Journal of Machine Learning Research 23 (2023) 1-62 Submitted 2/23; Revised X/Y; Published X/Y

Empirical Design in Reinforcement Learning

Andrew Patterson AP3@UALBERTA.CA
Samuel Neumann SFNEUMAN@UALBERTA.CA
Martha White WHITEM@UALBERTA.CA
Adam White AMW8@UALBERTA.CA
Department of Computing Science and Alberta Machine Intelligence Institute (Amii)
University of Alberta, Edmonton, Canada

Editor:

Abstract

Empirical design in reinforcement learning is no small task. Running good experiments requires attention to detail and at times significant computational resources. While compute resources available per dollar have continued to grow rapidly, so have the scale of typical experiments in reinforcement learning. It is now common to benchmark agents with millions of parameters against dozens of tasks, each using the equivalent of 30 days of experience. The scale of these experiments often conflict with the need for proper statistical evidence, especially when comparing algorithms. Recent studies have highlighted how popular algorithms are sensitive to hyper-parameter settings and implementation details, and that common empirical practice leads to weak statistical evidence (Machado et al., 2018; Henderson et al., 2018). Here we take this one step further.

This manuscript represents both a call to action, and a comprehensive resource for how to do good experiments in reinforcement learning. In particular, we cover: the statistical assumptions underlying common performance measures, how to properly characterize performance variation and stability, hypothesis testing, special considerations for comparing multiple agents, baseline and illustrative example construction, and how to deal with hyper-parameters and experimenter bias. Throughout we highlight common mistakes found in the literature and the statistical consequences of those in example experiments. The objective of this document is to provide answers on how we can use our unprecedented compute to do good science in reinforcement learning, as well as stay alert to potential pitfalls in our empirical design.

Keywords: Reinforcement Learning, Empirical Methodology

1 Reinforcement Learning, an Empirical Science

Running a good experiment is difficult in reinforcement learning. There are many decisions to be made. How long should you run your agent? Should you count the number of episodes or number of steps? Should performance be measured online, or off-line with test trials? How should you measure and aggregate performance? Do we use rules of thumb to set hyper-parameters or some systematic search? What are the right baseline algorithms to compare against? Which environments should you use? What does good learning even look like in a given environment? The answer to each question can greatly impact the credibility and utility of the result, ranging from insightful to down-right misleading.

The task of evaluating a reinforcement learning agent is complicated by the fundamental aspect that makes the problem interesting: an agent interacting with an environment. Our problems are not fixed datasets like in supervised learning. Reinforcement learning experiments are online and interactive: the agent—a program—generates its own training data by interacting with the environment—another program—and the quality of the data depends on what the agent previously learned. This interaction makes fair comparisons and scientific reproducibility major challenges in RL. There is no clear analogy to test and training splits that facilitate statistical claims in supervised learning. In fact, many of the ideas from classical machine learning such as overfitting, cross-validation, and model selection are either different or non-existent in RL. It is not surprising that the community is currently wrestling with the consequences of limited reproducibility, experimenter bias, unreliable algorithms, and exaggerated performance claims.

The field of reinforcement learning is experiencing rapid growth and many of the issues we see today are expected of a growing field. Historically, the community was much smaller than other branches of machine learning and the scale of most experiments was limited: considering a half dozen state dimensions and thousands of episodes represented large scale. Before 2015 (with the development of DQN), researchers were likely to begin by replicating experiments from the Sutton and Barto textbook (Sutton and Barto, 2018) and then extending and innovating from there. This was perhaps serendipitous as Sutton and Barto spent decades refining their experimental insights: learning from animal learning experiments and from teaching the textbook year after year. Most researchers were starting from an excellent empirical foundation. Historically, large scale experiments in RL such as TD-Gammon (Tesauro, 1995) and work in robotics were demonstrations highlighting what was *possible* with RL, without attempting to make strong scientific claims.

This document makes a distinction between scientific studies in RL and demonstrations of (impressive) engineered systems. Both play an important role in RL research, but can be detrimental when conflated. Demonstrations can be seen as exploratory science, probing the edges of what is known before following with a more thorough empirical study. They can also be focused on demonstrating the capabilities of existing algorithms, in hard problems or applications. Scientific studies, on the other hand, aim to obtain a deeper understanding of our systems and algorithms. As with any scientific study, there needs to be a clear hypothesis that is falsifiable and controls for confounding effects. **The aim should not be to show an algorithm is good, but rather understand an algorithm's properties, potentially relative to other algorithms.**[1]

Given the immense growth of the field, there is an emerging need to more clearly articulate and even develop better empirical practices in RL. There is certainly a greater variety of researchers operating in RL currently. Many come from other fields of machine learning and neuroscience, bringing with them different expectations, practices, and rules of thumb. Much of the widely bemoaned poor empirical practices (Henderson et al., 2018; Jordan et al., 2020; Agarwal et al., 2021; Colas et al., 2018) could be due to mistakenly applying practices common in other communities.

1. Of course, an important part of understanding is also theoretical analysis. Experiments and analysis go hand-in-hand, in that they both aim to provide understanding, using different tools. Sometimes the right answer is to use analysis, rather than experiments, or to do both. This document is focused on good experiments, but does not suggest that this is the only route to understanding our algorithms.

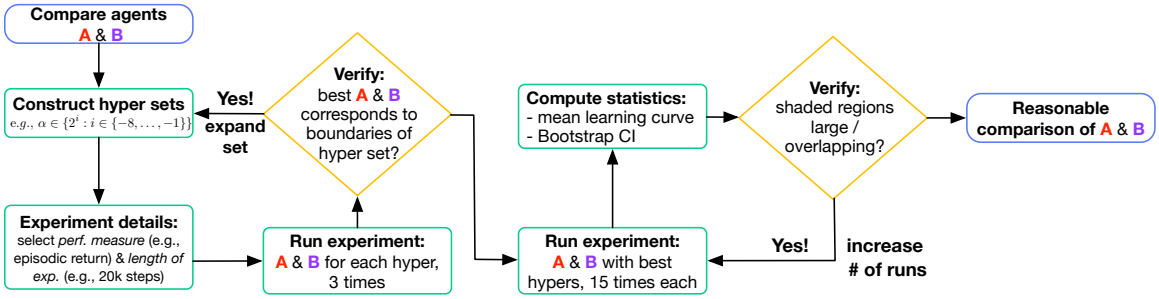


Figure 1: **Two-stage approach to comparing two algorithms.** The goal of this experimental workflow is to progress left to right. Good choices in the green boxes should limit the experiment rerunning sometimes forced by the yellow decision diamonds. Each stage of the work flow is discussed in detail in the text; we link each for convenience.
Section 2: discusses how to decide on key *experiment details (in 2.7)*, the basics on *how to run an experiment (in 2.1)*, plot *learning curves and confidence intervals (in 2.5)*, and decide if we *need more runs (in 2.6)*.
Section 3: discusses how to *construct hyper sets*, and how to determine if you need to *expand the hyper set (in 3.1)*.

In this document we aim to provide a cookbook or how-to guide for running good experiments in RL. We will walk the budding RL empiricist through important design decisions, common mistakes, and hidden biases. We will provide numerical examples of the consequences of bad decisions and illustrate what clear results and fair comparisons look like. In some cases we convey rules of thumb and sources of bias hard learned over decades of experience in the field. Naturally, we can never cover all the key decisions and the list of bad practice will be incomplete and ever growing. Regardless, our ambition is to provide (1) a reference on how to run good experiments in RL for those new to the field, and (2) additional insight and examples so that this cookbook may be useful to the seasoned researcher as well.

We begin by discussing the complexity of a first experiment one might run: evaluating a single agent on a single environment in Section 2. The section introduces key concepts about what to measure and how to aggregate performance in Section 2.1. Then we get more technical, looking at sources of variability in our experiments (Section 2.2), and how to make statistically significant claims (Sections 2.4, 2.5). In Section 3 we address how to deal with hyperparameters in our experiments. After that we move to comparing multiple agents, and the additional nuances that arise in Section 4. In Section 5 we discuss some considerations when selecting environments for your experiments. We conclude with a summary of common errors made in experiments in Section 6. Finally, in Section 7 we attempt to recreate a previous result and demonstrate how to use the strategies explained here to improve on the previous experiment design.

With such a large document, it is often useful to provide a high-level summary. We provide a flowchart visualizing an experimental procedure, with references to associated sections; we provide this upfront, so that the reader can come back to it as they read the details. We specifically visualize a common two-stage procedure, which is can be improved upon (as we discuss in Section 3.2), but nonetheless represents good empirical practice and should lead to reasonable conclusions.

This document is educational, but does in fact contain a variety of new results. These results are used to support methodological proposals throughout this work. For easier reference, we list those novel findings in Appendix A. We also include a summary list of common errors and pitfalls at the end of this document, in Section 6.

2 Observational experiments

We start with a simple observational study of a single agent interacting with an environment. Only once we have mastered the art of observation can we begin to design controlled experiments to understand the properties of our agents and their underlying learning systems. In an observational study, we—the experimenters—do not attempt to control the outcome of the study. A classic example is the "bakeoff"-style of experiment, where several baseline algorithms are tested across several benchmark environments. The results of such a study lay the groundwork for identifying questions of interest for further experimentation.

Because much experimentation and observation in RL occurs in deterministic simulation, we as researchers have a far greater degree of control in the design of our experiments than most natural sciences. This control can often become distracting and cause us to lose sight of our original goals as empiricists. As such, throughout this paper we will reference parallels to other fields of empirical science in order to provide grounding for our empirical practices and to build intuition.

Example:
Imagine an animal learning laboratory where we study how quickly rats can learn to navigate a maze. We place an individual rat in the maze and record some demographic details; its height, weight, age, sex, and so on. We then observe the rat's behavior as it begins to explore its environment and learns to obtain a block of cheese at the end of the maze. Throughout, we measure time-to-completion, the number of wrong turns taken, and some qualitative measures such as the perceived frustration of the rat throughout its learning process.

It is clear that some of the parameters of the individual rat will play a role in the time it takes for the rat to complete the maze. Highly fit rats will likely complete the maze faster than particularly lazy rats, though the degree of influence is complex and unknown. Further, these rats are all individuals of the same subspecies and differences between the individuals are unknown to the researcher and so are treated as random factors (i.e. the researcher does not observe the genetic make-up of each rat, nor does the researcher know each rat's detailed history of experiences before entering the lab). However, some elements of these rats are expected to remain consistent because they belong to the same subspecies—for example, this subspecies is known to be generally smaller and more docile than other subspecies of domesticated rats.

The field of reinforcement learning shares many similar motivations in both how and why we conduct experiments as in this example. Often, we evaluate the quality of a policy through the use of a value function; similar to measuring how quickly a given rat completes the maze task. However, this is only the first level of analysis. The goal in animal learning is not to understand how well rats can complete mazes, but rather to use the maze task to learn about the rat's ability to learn and explore.

The second analogy is in how we obtain repeated trials. We can describe several characteristics of a particular subspecies of rat—one subspecies is more docile, another generally larger—and these characteristics inform us about general behavior of that subspecies. However, within a subspecies there is still variation across individuals. For example, although lab rats are generally docile, we might still encounter a particularly ornery rat. To understand and characterize a subspecies, the researcher needs to average over these individuals. Likewise, we are interested in performance across multiple agents produced by a given algorithm, environment and experimental setup. In order to ascribe behaviors to a learning algorithm, we must first account for the individual differences of the agents it produces.

Let us establish some terminology before diving into our first set of experiments. Throughout this paper, we use the term **agent** to refer to a single entity interacting with an environment; analogous to an individual rat in a maze. We use the term **algorithm** to refer to a process which produces a set of agents, both by specifying initial conditions such as the initial weights of a neural network, and by specifying the learning rules by which the agent adapts to observations. Algorithms typically have configuration parameters, often called hyperparameters, which modify the set of individuals produced by the algorithm. We call an algorithm with a particular hyperparameter configuration a **fully-specified algorithm**, which is analogous to a particular subspecies of rat such as *Rattus norvegicus domestica*, the common lab rat. Finally, an **unspecified algorithm** refers to an algorithm where some (or all) hyperparameters have not been configured. This is analogous to a species of rat, such as *Rattus norvegicus*.

2.1 Experiment one: a demonstration

The first step towards designing an effective experiment is identifying the scientific question (or hypothesis) of interest. In this first section, we will focus on the simpler observational studies which typically seek to answer questions of the form "*How well does algorithm A perform on environment E?*" We will specifically be investigating the Expected SARSA (or ESARSA) algorithm on a simple maze gridworld environment. Details of the learning algorithm and environment can be found in Appendix B. For now it is sufficient to know that individual agents have sufficient learning capacity to efficiently to find a near optimal policy in this environment.

Now that we know which agents we want to observe and on what environment, there are still many design decisions to be made. For example: (1) how many time steps will each trial or run contain? (2) if a terminal state is not reached after n steps, will we artificially terminate the episode? (3) if the task is episodic, how many episodes should we run? In general these choices should be made based on what you want show. We will illustrate our thought process with an example experiment.

The environment is shown in Figure 2. The objective is to learn the shortest path to the goal over repeated episodes. The actions are discrete, moving the agent a fixed amount (plus noise) in the continuous two dimensional space. Actions that would move the agent outside the bounds of the world or into a wall cause no change in the state. The observation (and MDP state in this problem) is the x,y position of the agent. There is a fixed start state and goal region, and the reward is +1 for reaching the goal region, which ends the episode, and the reward is zero otherwise. The discount is $\gamma = 0.99$. Notice that, for this reward

specification, the discount needs to be less than 1 to encourage the agent to reach the goal quickly: otherwise, taking 100 or 1000 steps would result in the same episodic return. For this simple problem we designed, we know that the optimal policy can get to the goal in 15 steps, meaning the optimal episodic return is $0.99^{15} = 0.86$.

Let us start with a common and simple case: running ESARSA on our maze for a fixed budget of time-steps measuring online performance. The goal in this problem setting is to get high episodic returns, so the performance we report is the discounted return for an episode.[2] We use a fixed budget of agent-environment interactions, meaning we run ESARSA on the maze for k steps. If k is large enough, then the agent will reach a terminal state and begin a new episode many times. With a fixed budget of steps, the agent will complete a variable number of episodes in k steps depending on how good the agent is at the problem.

We use a fixed budget of steps to better reflect that we care about online performance, where each sample matters. It lets us ask: after n steps of environment interaction, how good is the agent's policy? If instead we use a fixed number of episodes, then some agents can get much more experience for learning. For example, a model-based agent could thoroughly explore in the first episode to learn its model, taking 9999 steps, whereas a model-free agent might find the goal in 100 steps. On the next episode, the model-based agent might already have a near optimal policy, looking like it learned much faster. In contrast, if we had considered the number of environment steps, then we might not find a big difference. It is not clear that our algorithms are designed to actually do this, but nonetheless we should measure what best reflects the goals of our experiment.

One secondary advantage is that it avoids highly variable runtimes. Under a fixed budget of episodes, poorly performing agents can have very long episodes, resulting in much longer runtimes. See Machado et al. (2018) for further discussion on this in the context of Atari.

Now let us consider how we can plot the learning curve for this agent, to see both how quickly it is learning as well as observe its final performance at the end of learning. To plot the return at time step t, we use the return for the current episode G_j that began on timestep $j \geq 0$,

$$G_j \doteq R_{j+1} + \gamma R_{j+2} + \ldots + \gamma^{T-1} R_{j+T},$$

where T is the (variable) length of the episode and $j \leq t \leq j + T$. The learning curve will be a piecewise linear step function, where the plotted performance will be the same for every step of the episode. For this particular environment, the rewards in this return are all zero except $R_{j+T} = 1$, making $G_j = \gamma^{T-1}$.

Figure 2 shows the result of our first simple experiment. We plot the return per step of our ESARSA Agent on the maze. The curve starts just below 0.2, because the agent randomly found the goal early in learning in under 200 steps (notice $\gamma^{200} = 0.13$). Then it is flat at zero for some time as its next episode takes so long that the return is effectively zero. The longer the flat portion, the longer it took the agent to complete the episode. The curve has a step-profile because of the way we plot episodic return versus steps and because the data only reflects the performance of a single run of the experiment.

2. Many deep RL algorithms are deployed with discounting even in undiscounted episodic cost-to-goal tasks. When this choice is made, it should be considered part of the internal mechanics of the agent: a hyperparameter inside the agent that induces additional discounting. This additional discount is not part of the environment and thus the performance metric should be the undiscounted return.

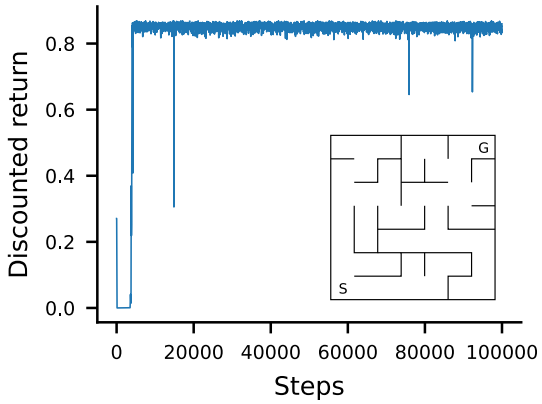


Figure 2: A single run of an Expected Sarsa agent on a simple maze problem. The return rate for this agent is $M = 0.827$. The agent has near optimal performance near the end of the curve, as it reliably reaches the goal in 15 to 17 steps, with the return hovering around $0.99^{17} = 0.84$ to $0.99^{15} = 0.86$.

It will also be useful to be able to summarize this performance over time. In this case, because we care about online performance—and so how much reward the agent receives while learning—it is natural to report the average over the points in the curve, or the sum (often called the area under the curve). We call this average over the learning curve the *return rate*. Such summary or aggregate performance numbers M are particularly useful when we want to compare agents (Section 4) or reason about quality of hyperparameters (Section 3).

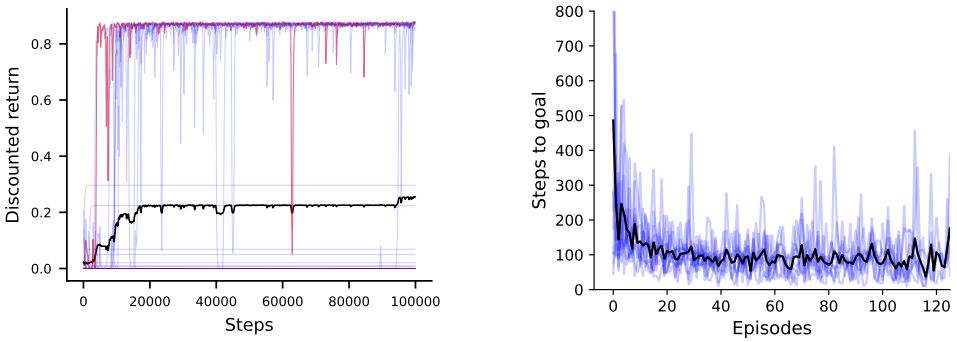
Remark 1

Other measures of performance and other aggregation functions for our learning curves can be considered. For simplicity in this work, we largely focus on measuring online episodic returns with this simple averaging aggregation. We discuss other choices in more depth in Appendix F.

We can see that this agent reached reasonable performance (near optimal) in this maze problem, though it is a bit hard to say if it learned quickly because we have no comparators. But, in any case, this was only one run! Perhaps we just got lucky? We need more independent evaluation of the agent to better characterize the performance.

2.2 Experiment two: characterizing variations in performance

Our first result was demonstrative in nature, however, most of the time we are interested in results that capture the reliability of our algorithms. Why would our algorithms perform differently if we ran them more than once? There are two primary sources of variation across agents produced by an algorithm; think *nature versus nurture*. The first source of variation we encounter occurs when we initialize a particular agent. Often, our function approximators—particularly neural networks—are randomly initialized causing differences from other similarly constructed agents even before data is observed. The second source of



(a) Performance of RL Agents in a Maze.

(b) Performance of Rats in a Maze

Figure 3: Understanding variability in agents. **(a)** 30 individual ESARSA agents on the simple maze environment. The thick blue line shows the mean over individual agents over time. **(b)** Raw data from 10 rats running a water maze.

variation comes from the data—the stochasticity in the environment—that is, what particular stream of data the agent observes throughout its life.

In Figure 3a we plot the best and worst performance of the ESARSA agent in the Simple Maze problem, highlighting a large variation in performance. The right hand side Figure 3b shows data from an animal learning experiment: 10 different rats running a maze on 10 different days.[3] It is interesting how much variation we see in the rat data. It should not be too surprising to us that learning agents, even using the same algorithm (same species), could exhibit quite a bit of variability.

Our efforts as RL empiricists are not that different than animal learning researchers and thus the motivations for repeated trials is also similar. We do not make claims about the abilities of a particular species of rat from an individual run though a maze. Each rat will be slightly different, due to random genetic factors and raising. Each time an individual is run through the maze things will be slightly different: lighting, humidity, the way the researcher holds the rat, etc. This maps pretty well onto the differences we see in agent initialization (e.g., neural network initialization) and environment variability (either simulated like sticky actions in Atari (Machado et al., 2018) or real-world issues like motor actuations).

2.3 Distributions matter!

There are many ways to characterize the variation in our agent's performance. It is common practice to report the sample mean and sample variance, though this can be problematic if the distribution is not normal. That's right, the variability in performance across agents gives rise to a distribution over performance for our algorithm. A single run of an experiment is a sample of an algorithm's performance on an environment. If we run an algorithm on a single environment multiple times, then we are repeatedly sampling from a distribution: $\mathbb{P}(M)$

3. It is common to exclude the data from poorly performing animals (after several retries). This practice has been adopted because scientists have identified genes that cause poor performance on particular tasks Vorhees and Williams (2006). In RL, we have no such prior knowledge which easily justifies removing data; dropping outliers should be done with caution.

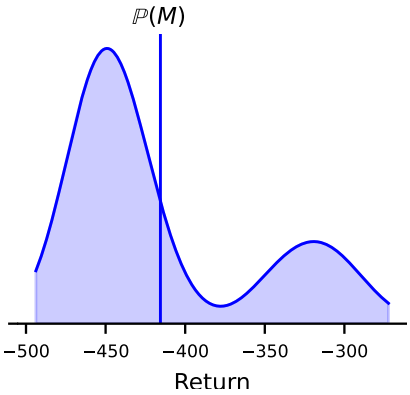


Figure 4: Performance distribution $\mathbb{P}(M)$ of DQN on Mountain Car with stepsize=2^{-9}. The performance M is the final performance of the agent after 100,000 steps of learning. The density around values for M represents the probability of an experimental trial yielding a given outcome. For instance, if we run DQN for a single random seed we will most likely observe an agent that achieves a return of approximately -450 by the end of learning. This density is estimated using 1000 independent DQN agents.

where M is the aggregate performance for a run. The stochasticity comes from changing the algorithm's random seed for initialization and decision making, and by changing the environment's random seed for simulated noise and start states. If $\mathbb{P}$ is skewed or bimodal, then we might need more than a dozen independent runs to estimate the mean and variance. Worse, these simple sample statistics might be misleading.

We can look at the skewed, multi-modal in Figure 4 to see why. The sample mean (black solid line) suggests the average agent would reach the goal in around 425 steps of interaction. However, a more complete description would be: any randomly selected agent is most likely going to require about 500 steps of interaction, while some agents will only require 250 steps.

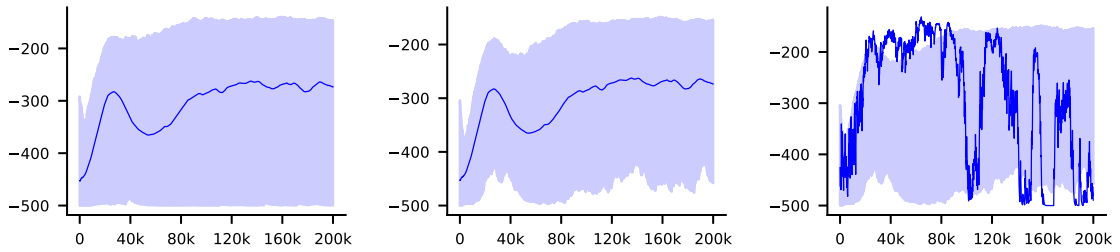
2.4 Reporting variability in performance

The performance distribution plotted above provides a richer summary of an algorithm's performance over many repetitions of the experiment, compared with say simply averaging learning curves (e.g., plot of episodic return vs time-steps). In general, we will not want to visualize this distribution for all algorithms; instead, we will want some summary statistics.

A more distribution-agnostic approach to summarize this curve is to provide upper and lower percentiles, (a, b), that reflect the range of performance. For example, we might want to know the lower .05 percentile a and upper 0.95 percentile b. Or, in other words, we might want to know the range that captures the center $\beta = 0.9$ percentage of the distribution. However, because we only have a small number of samples from this distribution, there is some uncertainty whether we have accurately captured the true shape of the distribution.

Tolerance intervals provide a distribution-agnostic way to summarize the range of an algorithm's performance while taking into account uncertainty due to a limited number of samples. We want to ensure that we capture this range with some confidence level $1-\alpha$ (e.g., $\alpha = 0.05$). To get a (α, β)-tolerance interval, the approach essentially involves computing the empirical upper and lower $(1-\beta)/2$ percentiles and then slightly widening the interval based on the numbers of runs used to compute the empirical percentiles. The formulas are simple, and standard computing packages include support for tolerance intervals; we provide more details in Appendix C.

Let us return to our experiment with DQN on Mountain Car and examine the variability. We ran DQN for 50 runs, varying the seed for the agent and environment together. We plot



(a) $\beta = 0.9$ with mean (b) $\beta = 0.7$ with mean (c) $\beta = 0.7$ with **median**

Figure 5: Tolerance intervals for the discounted return of DQN on Mountain Car, over 50 runs with $\alpha = 0.05$. Recall β specifies the percentage of the distribution considered for the tolerance interval. **(a)** Tolerance interval with $\beta = 0.9$, with mean performance. **(b)** Tolerance interval with $\beta = 0.7$, with mean performance. **(c)** Tolerance interval with $\beta = 0.7$, with **median performance**.

mean performance with an $(\alpha = 0.05, \beta = 0.9)$ tolerance interval in Figure 5a. The range in the plot is created by computing tolerance intervals for the 50 values at each time step t.

Finally, we might also want to visualize the performance of the *median* agent instead of the *mean* performance over agents. The mean performance does not correspond to any of the agents in the 50 runs. It might be useful to know how a representative (median) agent performed. We visualize this in Figure 5c. We take the average episodic return over learning in each run, to get $M_1, M_2, \ldots, M_{50}$, and pick the run j that is the median in $M_1, M_2, \ldots, M_{50}$. We plot the learning curve for this agent. We can see that this median agent exhibits considerably higher variation per timestep than the mean learning curve over agents. We also highlight that the tolerance interval is not centered around this median agent. Rather it is centered around the median over all agents taken at each timestep.

Standard deviations and tolerance intervals reflect the variation in performance. As we get more and more runs, the sample standard deviation and tolerance intervals approach their true values: the true standard deviation and the true probability interval that captures β proportion of the population. They do not shrink to zero with more runs, unlike confidence intervals, which we discuss next.

2.5 Reporting confidence in performance estimates

The majority of results in the RL literature report confidence intervals which are very different than what we have discussed so far. The mean and variance distribution plots and tolerance intervals attempt to capture the variation in the distribution of performance. As you collect more and more data—more runs—you get more and more accurate summaries of the variation. A confidence interval on the other hand has a different goal: to capture how certain we are in our estimate of some statistic about agent performance. As you collect more and more data we become more and more confident that our estimate has converged.

Confidence intervals allows us to report our uncertainty in the mean estimate, for example. We obtain a confidence interval (l, u) for a given confidence level $1 - \alpha$, where the interval is wider if we desire a higher confidence: wider for $\alpha = 0.01$ than $\alpha = 0.05$. The interpretation is that there is a low likelihood α that the true mean falls outside of (l, u). The uncertainty

comes from the finite sample we observe, meaning our interval could have been different had we seen a different sample (different runs). In other words, we can say that at least $1-\alpha$ percentage of the time with a different random sample, our interval would contain the true mean. It is possible that we were unlucky and have the finite sample where the true mean is not in our interval, but it is unlikely that we have that interval.[4]

To make such probabilistic statements, we have to make different assumptions about the underlying distribution over performance. The distribution could be Gaussian (normally distributed), but more generally could be any distribution, such as the bimodal, skewed distribution in Figure 4. We cannot know ahead of time exactly what our distribution looks like. But, there is a wealth of literature on selecting appropriate confidence interval approaches (see Japkowicz and Shah (2011) for a good reference).

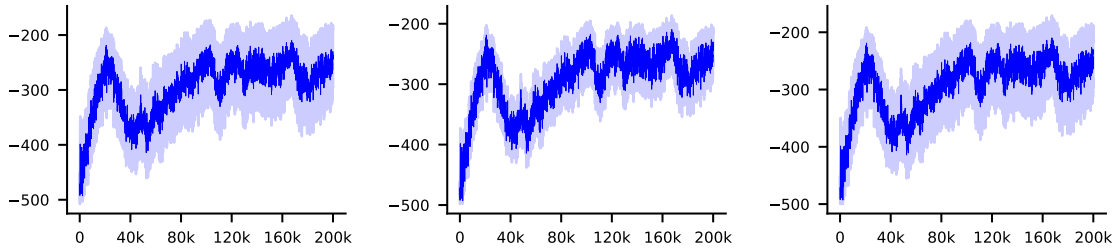
A reasonable choice is to obtain a confidence interval using the Student t-distribution. This choice assumes the underlying distribution is approximately Gaussian. A typical recommendation before using this approach is to visualize the empirical distribution over your samples, to see if normality is a reasonable assumption—a so called graphical method.[5] For example, we could plot the 50 sampled scalars $M_1, M_2, \ldots, M_{50}$ and see if they are concentrated around the mean value, or even use a package to plot an empirical distribution. More generically, if we assume we have n samples of performance, the Student t-distribution confidence interval is of the form

$$\left[\overline{M} - t_{\alpha,n}\frac{\hat{\sigma}}{\sqrt{n}}, \overline{M} + t_{\alpha,n}\frac{\hat{\sigma}}{\sqrt{n}}\right] \qquad \text{where } \overline{M} \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} M_i \quad \text{and } \hat{\sigma}^2 \stackrel{\text{def}}{=} \frac{1}{n-1}\sum_{i=1}^{n}(M_i - \overline{M})^2.$$

The multiplier $t_{\alpha,n}$ depends on the confidence level and the number of samples. For example, for $\alpha = 0.05$, as the number of samples increases, $t_{\alpha,n}$ gets closer to the typical 1.96 for Gaussian distributions.[6] This multiplier can be obtained from the Student t-distribution table, or again computed using standard computing packages. As an example, for $\alpha = 0.05$, with $n = 3$ (two degrees of freedom) we have $t_{\alpha,n} = 4.303$, for $n = 10$ we have $t_{\alpha,n} = 2.262$ and for $n = 1000$ we have $t_{\alpha,n} = 1.962$.

The confidence interval itself shrinks with more samples, because the standard error term $\frac{\hat{\sigma}}{\sqrt{n}}$ goes to zero. In our setting, this means as we get more and more runs, our confidence interval around our mean estimator $\overline{M}$ shrinks to zero until we can confidently claim that we have an accurate estimate of the mean.

4. Mean performance across runs is commonly reported in RL, and so we discuss appropriate empirical procedures here for estimating it. It is important, however, to reflect for yourself on when it is useful or not useful to use this evaluation. As we saw in the previous section, the mean performance is not necessarily reflective of any agent. The behavior of each agent can actually be quite erratic, but the mean performance might be reasonably smooth. In some settings, we want each agent to behave reasonably; reporting just the mean does not allow us to see potentially poor or erratic behavior in these agents. In other settings, such as when the environment is highly stochastic and the primary cause of variability, then it might be sensible to report just the mean performance.
5. Alternatively, one could use a goodness of fit test like the Kolmogorov–Smirnov test or other statistical techniques. Many software packages such as Matlab, R, and SPSS have implementations you can use.
6. Using a Gaussian confidence interval requires knowing the true variance. Because we have to estimate it from data, we actually have uncertainty in this part of our interval as well. Therefore, our interval is actually a bit wider than if we knew the true variance. With more samples, our estimate of the true variance (true standard deviation) becomes accurate and so the multiplier approach the same multiplier we get if we had the true variance.



(a) $\alpha = 0.05$ with Student's t (b) $\alpha = 0.3$ with Student's t (c) $\alpha = 0.05$ with bootstrap

Figure 6: Confidence intervals around the discounted return of DQN on Mountain Car averaged over 30 runs. **(a)** Student's t-distribution confidence interval with $\alpha = 0.05$. **(b)** Student's t-distribution confidence interval with $\alpha = 0.3$. **(c)** Bootstrap confidence interval with $\alpha = 0.05$, with mean performance. With 200 runs the mean curve is much smoother and the CIs are tighter. We choose 30 runs for illustration purposes.

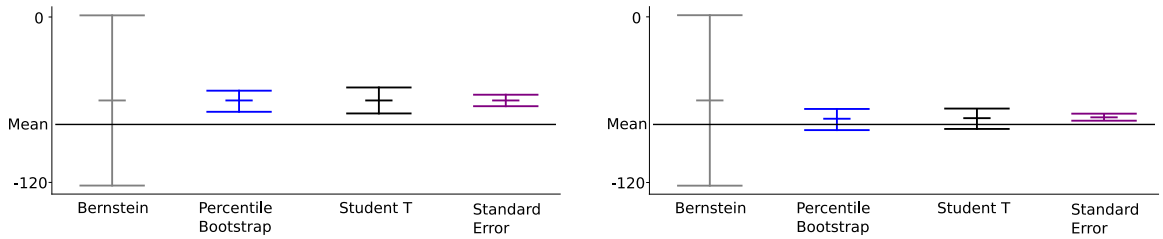
We run our third experiment, where we compute a confidence interval around the mean estimator—instead of a tolerance interval—using the same data as in the last section. We can see in Figure 6a that the confidence interval with $\alpha = 0.05$ is already quite tight, for these 50 runs. This shaded region reflects our uncertainty in our estimate of the mean, whereas the tolerance interval in Figure 5a reflects the variation around the mean (and so is wider and does not shrink to zero).

We can ask how many samples n we need before it is reasonable to compute this confidence interval. We can actually obtain a statistically valid confidence interval, even with only two samples! The interval itself will simply be wider, because $t_{\alpha,n}$ will be larger as will the standard error. However, it is difficult to gauge, with only two samples, if it is appropriate to make the assumption that the underlying distribution is Gaussian. You need to obtain enough runs to decide if the Student t-distribution confidence interval is appropriate.[7]

In many cases we may not be confident that our performance distribution is Gaussian, or we might even believe it is not Gaussian. In that setting, a natural alternative to the Student's t confidence interval is to use bootstrap-based statistics to generate confidence intervals. The bootstrap procedure is simple. As before, we get n measures of performance for an algorithm, using n random seeds. We then generate a new dataset by resampling n values from the original dataset *with replacement*. Finally, we repeat this resampling procedure for a total of m times, with m usually very large (e.g. $m = 10,000$). For each of these new datasets, we compute the statistic of interest (e.g. the mean) then measure the variability of that statistic over all of the m datasets. To create a 95% confidence interval, we report the 0.025 percentile of the estimated statistic over all m estimates as a lower-bound and the 0.975 percentile as an upper-bound.

A major advantage of bootstrap-based methods is that they often require very few assumptions about the underlying data. This advantage comes at a cost, however, because bootstrap methods generally require more data points to provide tight confidence intervals. For this reason, our recommendation is to default to bootstrap-based methods for most comparisons, but to check the underlying distributions to see if more powerful methods—such as Student t-distribution confidence intervals—can apply without breaking assumptions. We visualize the bootstrap confidence interval using the above procedure in Figure 6c.

7. A common rule of thumb is that 30 runs is enough. There is no clear justification for this number.



(a) Confidence intervals with 10 runs.

(b) Confidence intervals with 50 runs.

Figure 7: Confidence intervals about the mean total discounted return of DQN on the PuddleWorld environment. Long-tail behavior in the performance distribution violates assumptions for the Student-T distribution, whereas the percentile bootstrap fails when too few samples are used. Increasing the number of samples can resolves the issue for the percentile bootstrap. All of these intervals should be 95% confidence intervals, except for the standard error. We include the standard error only to highlight that it can be much narrower than the corresponding 95% confidence interval. Implicitly, using a standard error is like using a Student-t with a low confidence level. In this instance, for $n = 10$, the standard error represents a Student-t confidence interval with $\alpha = 0.5$ (because $t_{\alpha=0.5,n=10} = 1$) and for $n = 50$, we have $\alpha = 0.3$ ($t_{\alpha=0.3,n=50} = 1$).

Remark 2

We did not seem to have to be so careful for tolerance intervals. But there too we had to account for uncertainty in our percentile estimates. The same distributional question arise. If we know we have an underlying Gaussian distribution, then we can get a better estimate of these percentiles with fewer runs. It is common, however, to default to distribution-free tolerance interval calculations, that make few assumptions about the data. We discuss this further in Appendix C.

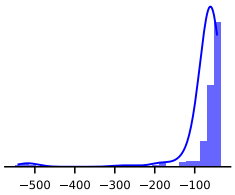
2.6 Do we really need more runs?

Nobody wants to run their experiment longer than needed. Just increasing the number of runs makes our experiments take longer, and has real environmental costs. Ideally, one would hope that advanced statistical tools can save us. Unfortunately, there is an inherent trade-off between making some assumptions and having tighter confidence intervals, and avoiding assumptions and having potentially useless confidence intervals. In this section, we highlight that we may need to do more runs, especially for the types of agents that we currently analyze that can have highly skewed performance distributions.

Consider the following example. Imagine we have collected 10 runs of the DQN algorithm on the PuddleWorld domain and we wish to report DQN's average performance. We can compute a confidence interval around this mean estimate, to reflect our uncertainty due to only having 10 runs. There are several different assumptions we could make, resulting in different confidence intervals. We visualize several options in Figure 7. In this synthetic example, we can actually compute the true mean and check: did our confidence intervals capture the mean? We can see the estimated average performance for these 10 runs is

far from the true average performance of DQN in this environment. Further, four of the confidence intervals provide overly optimistic ranges and fail to capture the true mean.

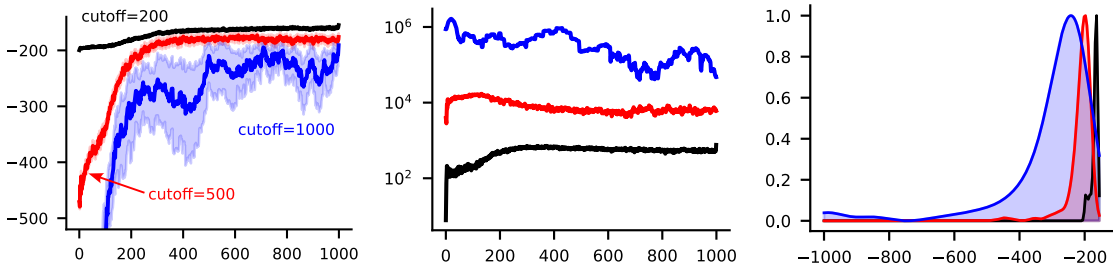
The sample Bernstein confidence interval does reasonably capture the population mean in Figure 7; a natural conclusion might be to prefer such confidence intervals which make minimal distributional assumptions. However the Bernstein bound is highly conservative, meaning provided confidence intervals are incredibly wide even for large numbers of runs. Unfortunately, this inhibits truly understanding whether we have accurately captured the average performance—in the case of PuddleWorld, the confidence region covers a majority of the range of possible means—requiring substantially more runs to present statistically meaningful results. In the example above, we required as many as 1000 runs to detect differences between DQN and an alternative algorithm using the sample Bernstein confidence interval, while the percentile bootstrap required only 30 runs.



Unfortunately, these examples where confidence intervals fail are far from rare. To see why they fail in this instance, we need only look at the performance distribution of DQN on PuddleWorld. The distribution is long-tailed with some very low performing runs occurring with low probability (approximately 5% of the time). In this case, all 10 of our samples were high performing causing us to overestimate the average performance while simultaneously underestimating the variation in performance. Note the probability of receiving 10 samples near the right-side mode is approximately 60%—not at all uncommon. The only way to resolve this issue is simply to collect more runs of DQN. For this specific example, we found that approximately 20 runs were sufficient to accurately reflect the high variation and obtain accurate bootstrap confidence intervals, and 30 runs were sufficient to accurately estimate the average performance.

One proposed solution to allow for a smaller number of runs, knowing that we have such distributions, is to instead report an estimate of the interquartile-mean (IQM) (Agarwal et al., 2021). The IQM takes the mean of the points in the interquartile range, namely between the 0.25 and 0.75 percentiles. This statistic is more robust to outliers, because these potentially larger magnitude values are not included in the mean calculation. The proposed estimate of the IQM is to drop the 25% highest samples (runs) and 25% lowest samples before computing the mean of the remaining 50% of the data (Agarwal et al., 2021).

As with any choice, we need to be cautious about whether the goal was to estimate the IQM, or whether we chose it because it provides a convenient way to reduce the number of runs. In some cases, it is a useful statistic, with the added benefit of being a robust statistic. In other cases, it may not capture key properties of the algorithm that we care about. Take the DQN algorithm as example. In our tested domains, we observed low-probability catastrophic failure events for DQN across nearly every tested domain. In Lunar Lander, some agents would simply fly off into oblivion, obtaining incredible amounts of negative reward until the episode was mercifully terminated due to episode cutoffs. In Cliff World, some DQN agents would get stuck in a corner perpetually in every single episode, never learning to find the goal. Even worse, a small subset of agents would learn to always jump into the cliff immediately and obtain massive negative rewards. In this case, removing these outlier agents—the agents whose performance do not conform to our pre-existing notions of how DQN *should* behave on simple domains—is not helping to create a clearer picture of our algorithm, as we are simply ignoring its shortcomings.



(a) Mean performance. (b) Variance over runs. (c) Distribution over runs.

Figure 8: Performance of the EQRC algorithm in Mountain Car for three different episode cutoff lengths: $\tau \in \{200, 500, 10000\}$. Smaller cutoffs clip the impact of outlier performance values, causing variance to be significantly under-reported and mean performance to be biased toward higher performance.

2.7 Deciding on the length of an experiment

An important empirical choice for understanding your agent is the number of steps of interaction. If you choose a smaller number of steps, then you are evaluating early learning performance. If you choose a larger number of steps, then you are evaluating if the agent can reach near-optimal performance within a reasonable number of samples. If you choose a very large number of steps, then you may be evaluating if your agent can reach near-optimal performance and stably remain at this performance.

One way to gauge if the number of steps reflects early learning or final performance is to examine the learning curve itself. If the curve is still increasing, then likely you are seeing early learning. If it has flattened for many steps, then you are seeing the final performance of the agent. To appropriately pick the number, you can run the agent for longer to gauge if it eventually levels off. Then you can report performance for a smaller number of steps to focus on early learning.

An additional choice related to steps of interaction is artificial episode cutoffs. Cutoffs are used to prevent an episode from becoming too long. An agent may get stuck, never finding its way to the goal and so never terminating the episode. A cutoff involves teleporting the agent after `max_steps` back to the start state, *without termination*. It is like picking up a robot that got stuck in a corner, and moving it back into a state from which it can learn.

For this reason, cutoffs can interact with exploration, and in particular very aggressive short cutoffs may misrepresent the performance of agents that get frequently stuck. For example, in original implementations of Mountain Car, no early cutoffs were used and tile-coded agents would run for a few thousand steps in the first episode on average (Sutton and Barto, 2018). In later implementations, an aggressive episode cutoff of 200 steps was introduced (Brockman et al., 2016). We show in Figure 8 that performance for an algorithm that uses neural networks—called EQRC (Patterson et al., 2022b)—is much better with more aggressive cutoffs, with much lower variability over runs.[8]

8. This change in default implementation illustrates the practice of problem adaptation—making an easier version of the problem to alleviate challenges in modern algorithms, such as neural networks struggling to learn from flat reward signals.

We might choose to run the agent for a fixed number of steps, rather than a fixed number of episodes. In that case, we do not need episode cutoffs; however, they can still be beneficial to use. The main reason is that, due to stochasticity in environments, many agents may have some interactions that lead the agent to get stuck. This issue may not only have to do with the agent, but also potentially with the environment. We can set the episode cutoff to a large number, that is less than the number of learning steps, to avoid having runs where the agent is stuck in one episode forever. With a large cutoff, we are much less likely to make the problem too easy and can avoid significantly skewing the results. But, we obtain some of the reduced variance in runs to facilitate evaluation.

Summary We conclude each section with a summary of key actionable advice. Here, we summarize key points to consider when investigating an algorithm in one environment, assuming we have already specified the hyperparameters.

Key insights: evaluating a fully-specified algorithm

1. A single run of an agent can be informative; consider visualizing individual runs to gain some understanding of your algorithm.
2. Agent behavior, with the same algorithm and environment, can be very different across random seeds. Consider reporting this variability using sample standard deviations or tolerance intervals, rather than just reporting the mean or median performance.
3. Confidence intervals around the mean primarily tell us about our certainty in our mean estimate, not about the variability in the underlying agents. If you primarily care about understanding mean performance, then report means with confidence intervals. If you care about the behavior of each agent, not just the mean, then consider reporting tolerance intervals.
4. **In general we advocate that you do *not* report standard errors.** They are like a low-confidence confidence interval, and it is more sensible to decide on the confidence interval you want to report.
5. The required number of runs depends on your performance distribution, which is unknown to you. It is clear that in almost all cases 5 runs is insufficient to make strong claims, but even 30 runs can be insufficient if the distribution is heavily skewed.
6. Consider reporting Performance versus Steps of Interaction, rather than Performance versus Episodes. This choice ensures every agent receives the same number of samples. (See Section 2.1)
7. Deliberately choose the number of steps of interaction to reflect early learning or ability to learn the optimal policy, rather than inheriting what was done in previous work.
8. Aggressive episode cutoffs can significantly skew the results.

There are lot of things to worry about just to run a good observational study of single agent. But let's gain some perspective here: why should this be easy? It's not easy for our animal learning scientist to conduct and document the study of a few rats in a maze. They

have to worry about so many other things that we do not: how to physically handle the animals, if their personal scent impacts the animals, uncontrolled genetic variations, running in real time, extra dull rats, and so on. In contrast, we can carefully enumerate our sources of bias, programatically vary conditions, exercise perfectly repeatable interventions, control almost all relevant sources of variation, perfectly record all relevant information, and run millions of experiments hundreds of times faster than real-time. Scientists examining the real-world cannot cut corners, even though their empirical setting is more onerous than ours. Similarly, if we truly want to understand our algorithms and gather sufficient evidence for our claims, we need to take the scientific enterprise seriously. Science is first and foremost about understanding, not picking winners and losers.

3 Dealing with Hyperparameters

Almost all algorithms have hyperparameters. These are scalars that have to be selected by a person before running an experiment. Typical hyperparameters in reinforcement learning include stepsizes and other optimization parameters like momentum and batch sizes; the eligibility trace parameter or the horizon for n-step methods; the target net refresh rate; and even the function approximation architectures used, which themselves can have many different hyperparameters (e.g., depth, number of nodes per layer, activation function, etc).[9]

The possible combinations of hyperparameters can be overwhelming. It is hard enough to properly evaluate an algorithm for a single hyperparameter combination, let alone having to consider this combinatorial space of algorithms. But, we can overcome this panic by stepping back and clarifying the goal of our experiment. There are three typical settings: (1) where we want to *understand the variety of behaviors* produced by our algorithms for different hyperparameter settings, (2) where hyperparameters are optimized in order to *understand the idealized maximum capabilities* of an algorithm, and (3) where hyperparameters are selected to mimic a deployment scenario in order to *understand the deployment performance* that the algorithm is likely to achieve. As a field, we generally have more understanding about how to study hyperparameter sensitivity (corresponding to setting 1), though of course empirical design here is also nuanced; we discuss this in Section 3.1.

This second setting falls into the category of competitive machine learning. Given the ability to extensively tune the performance of a learning system, what is the maximum capability we can hope to achieve? Such studies are limited to specific problem settings, often called benchmark problems. Unfortunately, there are multiple challenges that arise in this setting. The primary challenge is statistical: it is hard to estimate the maximum value of a stochastic function, such as the maximum performance of a reinforcement learning system. Another challenge arises in making fair comparison to baselines in the competition, it is difficult to ensure equal tuning effort is given to each competitor learning system. We discuss this issue further in Section 3.2.

The final setting poses the greatest challenge. Unlike supervised learning, we do not have a general purpose approach to select hyperparameters. In supervised learning, cross-validation

9. It is reasonable to expect that some of these hyperparameters—like learnable parameters—should adapt with time. However, we can consider hyperparameter adaptation as part of the algorithm; our job is to specify the hyperparameters for that adaptive algorithm. For this reason, we define hyperparameters to initial values set at the beginning of the experiment.

can be used with almost any algorithm to select hyperparameters according to generalization performance. It is not obvious how to use cross-validation in reinforcement learning (see a more in-depth discussion in Appendix D.3). An additional challenge comes from the variety of use cases for reinforcement learning, such as solving simulated problems versus interacting with the real-world; having access to lots of data but limited compute versus lots of compute and limited data; or settings where taking exploratory actions is safe versus unsafe. Each combination of these (and other) factors will lead to different methodologies to select hyperparameters for a reinforcement learning algorithm. Understanding the performance of an algorithm in deployment depends not only on the algorithm and the hyperparameters tested, but also on the deployment scenario itself. We discuss this issue further in Section 3.3.

In order to more precisely discuss the role of hyperparameters, we borrow some terminology from recent work on hyperparameter tuning for reinforcement learning (Jordan et al., 2020). Modern RL algorithms can have dozens of hyperparameters that need to be set before a single experiment can be run. Fortunately, the literature (and codebases) contains some reasonable default choices for many hyperparameters, significantly narrowing the space of unknowns that need to be specified before running an experiment. In the case that all of the hyperparameters have already been set—either through tuning, defaults, or an adaptive algorithm—we will call this a *fully-specified algorithm*. A fully-specified algorithm has no unknowns besides its learnable parameters and is ready to run on a problem setting. We will refer to algorithms that have at least one unspecified hyperparameter as *partially-specified algorithms*. It is in this partially-specified setting where we will start, with just one hyperparameter unspecified.

3.1 Understanding Hyperparameter Sensitivity

The goal of hyperparameter sensitivity analysis is to help us understand our algorithms. *This is not about optimizing hyperparameters to support SOTA claims!* These insights can help identify serious sensitivities that suggest improvements to the algorithm are needed, they can help us understand changes in behavior as we interpolate across a space of different algorithms, or they can help identify hyperparameters which require joint tuning in order to provide good performance. As a classic example, the trace parameter λ in TD(λ) algorithms interpolates between Monte Carlo algorithms as $\lambda \to 1$ and the original TD algorithm as $\lambda \to 0$. A sensitivity study may reveal that $\lambda = 0.9$ is an optimal choice for an environment; however, this is only one useful piece of information that we can derive from the study. We may learn that performance becomes highly variable as $\lambda \to 1$ or that λ near 0 diverges. We might additionally learn that the performance suddenly drops off outside a narrow range of λ suggesting that this algorithm will be difficult to tune on future environments.

In order to evaluate our algorithm with different values of a hyperparameter, we need to collect enough data to provide reasonable estimates of our statistic of interest. Say we wish to report the average performance across agents produced by our algorithm for each hyperparameter value, then for every hyper-parameter setting we need enough agents to actually estimate that average. This is no different than Section 2.1 where we required multiple agents to evaluate an algorithm, except now we are evaluating multiple fully-specified algorithms; one for each setting of the hyperparameter of interest. In the most basic setting, this means we need N runs for every hyperparameter setting, H, for a total of $N \times H$ runs. Clearly, this can become expensive quickly!

Dealing with a single hyperparameter. Once we have obtained an estimate of performance for each setting of our hyperparameter, we can summarize the performance of this partially-specified algorithm as in Figure 9. To create such a plot, we must first decide on a range for the hyperparameter then specify how intermediate values are selected within that range. A common range for stepsizes is to use powers of 2, to systematically cover the space. If there is a clear bowl or U-shape to the resulting curve—as in Figure 9—then this selection scheme was likely appropriate—though the curve is not always U-shaped. If we observe sharp changes in performance—like the V-shape in Figure 10—you may need to sample more densely within that region to better understand the range of appropriate values for the hyperparameter. These sharp changes often occur when the initial range of the hyperparameter is too large or when the distribution of tested values are concentrated around a region of poor performance—that is you missed the good ones. If the best performance is at the one end of the range—like in Figure 11— then this suggests the range was too narrow and may need to be systematically expanded.

Now that we have our sensitivity curve, how do we interpret it? If the sensitivity curve is reasonably flat—the minimum performance is close to the maximum—for a wide range of hyperparameter values, then we might say that this partially-specified algorithm is insensitive and so it will not be challenging to define a fully-specified algorithm for deployment. If the sensitivity curve indicates a large difference in performance within a narrow region of hyperparameter values, then we would say this partially-specified algorithm is highly sensitive and conclude that defining a fully-specified version for deployment might be difficult.

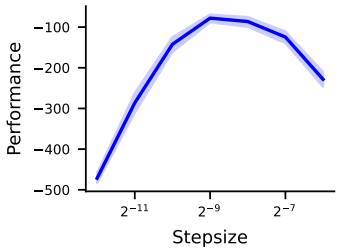


Figure 9: A good sensitivity curve that captures a wide range of the variable of interest and illustrates that performance changes smoothly as the hyperparameter changes.

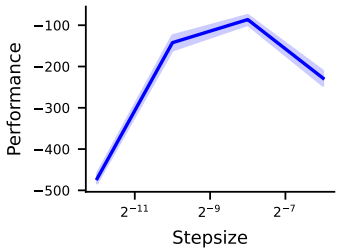


Figure 10: A sensitivity curve where the range of tested values may be too wide, instead of being focused in the region of interest. We lose some information around the peak performance and the algorithm appears quite sensitive. This sensitivity might be an artefact of the plot—testing insufficiently many values—rather than a property of the algorithm.

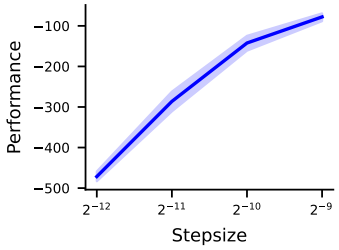


Figure 11: A sensitivity curve where we potentially missed the best performance. The best performing hyperparameter may be outside the range or may be the endpoint of the range, but we cannot tell with the presented information.

Such comprehensive experiments can be expensive, and it can be tempting to test only a small number of hyperparameter settings. However, we always have to ask ourselves if we have compromised our empirical design. **There is no point in running a flawed experiment, even if it is more feasible in terms of computation.** An alternative choice would be to ask research questions that better match our computational resources. Remember, some empirical questions are simply more challenging to answer than others. For example, it is challenging to convincingly show that an algorithm is insensitive—this would require testing a wide-range of values with a large number of intermediate values[10]—or that an algorithm is divergent.

Example: Does TD diverge?

We know that TD can diverge under off-policy updating. In fact, we can prove that TD always diverges in very specific scenarios such as Baird's counterexample (Baird, 1995). But it may not always be possible to theoretical characterize and algorithms convergence behavior theoretically. It is in these scenarios that we often rely on empirical evidence.

For example, imagine you combine TD's update rule with a momentum term. Does this still diverge on Baird's counterexample? If there is as yet no theory about convergence or divergence for TD with momentum, you might turn to experiments to obtain some insights. Likely, we would start with the default hyperparameters for momentum (say setting the momentum term $\beta = 0.9$). We find that this algorithm diverges! We test again with a different random seed and observe divergence; and again for another random seed, and again, and again... After many such trials, we might conclude with high confidence that TD with momentum always diverges on this problem setting. This conclusion, however, is limited by the choice $\beta = 0.9$. Does this conclusion hold for $\beta = 0.99$ or $\beta = 0.1$?

To provide evidence that TD with momentum does not converge in this problem setting—regardless of hyperparameter setting—we must carefully sweep a dense set of values of β and show divergence for every tested value. Without theory—or an infinite number of runs—we can never know with absolute certainty that TD with momentum always diverges. It is always possible that some untested random seed presents the data in exactly the right order such that TD with momentum converges, or possibly there is some perfect value of β where TD converges on this problem setting. With empiricism, we accumulate a body of evidence which supports our claim; the more random seeds we test and the more hyperparameter values we sweep, the more convincing our body of evidence.

Assessing overall hyperparameter sensitivity One of the primary goals of hyperparameter sensitivity studies is to understand how improve our algorithms and develop those that are easier to tune. In addition to explicitly visualizing performance versus a specific hyperparameter, we can also attempt to assess overall how sensitive an algorithm is to its hyperparameters. One approach (Jordan et al., 2020) is to treat all hyperparameters as unknown values. We model these unknowns as random variables and draw sample hyperpa-

10. We could more thoughtfully sample the hyperparameters to get an accurate sensitivity curve. One could imagine an automatic procedure that starts broad, and iteratively samples hyperparameter choices between two where the values are very different, or focuses on regions of higher performance where we want the curve to be more accurate. Bayesian optimization approaches provide such a strategy to find the best hyperparameters, and so the ideas there could potentially be adapted for this goal. Nonetheless, such an algorithm would still require a large number of hyperparameters.

rameter configurations. For example, we might treat the stepsize, α, as an unknown value and sample $\alpha \sim \text{Uniform}(0, 1)$. Each sampled α produces a fully-specified algorithm, for which we can obtain a measurement of performance. The amount that the performance changes as we change the hyperparameters then provides a measurement for the sensitivity of the under-specified algorithm on a given problem setting.

This procedure is akin to performing a sensitivity study across all hyperparameters simultaneously. Because the space of hyperparameters is combinatorial, exhaustively sweeping across configurations to obtain a measure of sensitivity is typically impractical. Instead, this approach randomly samples points in the combinatorial space in order to compute the variation in performance over that space. This presents a tradeoff: we can carefully and systematically investigate a small number of hyperparameters at a time, or we can try to investigate the entire space of hyperparameters with much less detail.

Remark 3

Dealing with hyperparameters is an active area of research and thus our treatment of it here is necessarily limited. The interested reader can jump to Appendix D.1 for an expanded discussion on handling multiple hyperparameters.

3.2 Reporting Idealized Performance

When we first introduce an algorithm, we may want to know how well it *can* perform, on an environment. If it performs poorly on an environment, even with well-tuned hyperparameters for that problem, then there may be an issue with the algorithm. Further, the algorithm may be conceptually appealing, but have new hyperparameters that have not yet been well-studied. If an argument can be made that adaptive algorithms could be developed for these new hyperparameters, then it can be appropriate to evaluate the behavior of the algorithm under nearly-optimal hyperparameter settings.

Though reporting performance (or behavior) for nearly-optimal hyperparameters is common in reinforcement learning, it has several serious pitfalls that you need to consider if you take this route. The first difficulty is that estimating the maximum of a stochastic function is challenging and often requires a very large number of samples. In our experiments, this translates to needing many random seeds and significant computational resources. The second is that many reinforcement learning algorithms are notoriously sensitive to their hyperparameters and often contain dozens of hyperparameters to tune. Finally, we have to be careful about comparing two algorithms under this idealized setting, where hyperparameters are optimized. If we allow one algorithm to have more hyperparameter settings, then differences in performance can be due to maximizing over more hyperparameter settings rather than differences in the algorithm. We discuss these issues in more details below.

Maximization bias. Several statistical challenges present themselves when tuning hyperparameters for maximum performance. One challenge is *maximization bias*, which is a form of statistical bias that can arise when estimating a quantity of the form $\max_h \mathbb{E}[G \mid h]$. In our case, h represents a hyperparameter configuration and G represents the performance of our algorithm for a given configuration. Because we do not know $\mathbb{E}[G \mid h]$, the average performance of our algorithm for a given hyperparameter configuration, we estimate it with

samples, $\bar{G}_h \approx \mathbb{E}[G \mid h]$. Then we estimate $\max_h \mathbb{E}[G \mid h]$ using $\max_h \bar{G}_h$. This estimate if prone to maximization bias, because $\mathbb{E}[\max_h \bar{G}_h] \geq \max_h \mathbb{E}[\bar{G}_h] = \max_h \mathbb{E}[G \mid h]$.

To gain a bit more intuition, consider the following example comparing two hyperparameter configurations h_1, h_2. Imagine that $\mathbb{E}[G \mid h_1] = \mathbb{E}[G \mid h_2]$, but when we estimate their average performance using samples, $\bar{G}_{h_1}$ is an overestimate of $\mathbb{E}[G \mid h_1]$. When we maximize over our estimates $\bar{G}_{h_i}$, we will be reporting an overestimate for $\max_h \mathbb{E}[G \mid h]$ meaning we are overstating the performance of our algorithm. Now imagine instead of two configurations that are identical, we instead have 100 identical configurations, or 1000. The probability that we overestimate at least one configuration increases as we increase the number of configurations estimated; thus increasing the reported performance for the algorithm, $\max_h \mathbb{E}[G \mid h]$. That is, without changing the properties of the investigated algorithm itself, we can report increasingly higher performance by increasing the number of hyperparameter configurations we investigate!

In fact, the above example occurs frequently even in simple reinforcement learning experiments. If we run 1000 experiments of DQN on the Mountain Car domain and sweep stepsizes, target network refresh rates, and replay buffer sizes for every experiment, then approximately 96% of these experiments will over report the average performance for the best hyperparameter configuration among those tested. As a result, we no longer have an accurate assessment of the performance of DQN on Mountain Car. If this was an algorithm that we are proposing, then we would be doing our readers a disservice by overstating the benefits of the algorithm. If this was a baseline algorithm, we would be doing ourselves a disservice by setting too high of standards and potentially filtering out useful ideas.

Issues with the typical two-stage approach. One common approach to counteract the effects of maximization bias is to use a two-stage methodology. In the first stage, the researcher performs an extensive hyperparameter sweep in order to select the maximizing hyperparameters. In the second stage, the researcher then evaluates the best hyperparameter configuration with a new set of random seeds—typically far more random seeds than originally used for the selection process. Finally, we report the average performance for the second stage only. This approach provides an unbiased, confident estimate of performance for that chosen hyperparameter configuration, because we are able to use more runs for that single hyperparameter configuration.

However, there are two key drawbacks to this approach. As pointed out by Jordan et al. (2020), this approach is wasteful of compute, because many samples of performance are thrown out during the hyperparameter selection stage. Additionally, this two-stage approach ignores uncertainty during hyperparameter selection. Say, for example, that we use 10 runs for every hyperparameter configuration in order to tune the algorithm. Due to maximization bias, however, we may unintentionally select a suboptimal hyperparameter configuration as being best. Then in the second stage, we will use many runs—say 100—to evaluate this hyperparameter configuration. Because we used so many runs in this second stage, our confidence intervals about the mean performance will be tight—we will be reasonably certain we have accurately captured the mean performance. This certainty, however, is unwarranted. While we may have accurately estimated the mean for the selected hyperparameter configuration, we likely have not accurately estimated the mean for the *best*

hyperparameter configuration. As a result, this two-stage approach is likely to underestimate the maximum performance and is overconfident in its estimate.

A new approach to obtain unbiased estimates of maximum performance. One way to overcome maximization bias is to repeat the experiment multiple times. Consider if after running the sweep over hyperparameters the first time, we repeated this procedure a second time using a new set of random seeds. We will almost certainly obtain a new estimate for maximal performance and likely obtain a different best hyperparameter configuration that gives that maximal performance. We now have two different estimates of maximal performance, giving us a sense of spread—how much does this maximizing performance change if we use different data to estimate it? Naturally, we can repeat this process many times—say 100—and report the mean and spread of the results.

Unfortunately, this simple procedure has a major downside: it is incredibly expensive. For every hyperparameter configuration (say H configurations), we require N runs and we repeat the sweep-then-maximize procedure M times. In total, we run our learning algorithm $H \times M \times N$ times in order to report an estimate of tuned performance.

Fortunately, we can rely on the principles of bootstrapping and resampling in order to make this procedure much cheaper. For every hyperparameter configuration, we collect N runs of our algorithm. Then we sample (with replacement) N of those N runs for every configuration. We compute sample averages for every configuration, then select the maximizing sample average as an estimate of tuned performance. Then we repeat the above procedure by resampling from the same set of $H \times N$ runs. Because we are capturing the variance across all hyperparameters, we do not need as many runs *per* hyperparameter. As a result, we can select a smaller N than we would typically require for sensitivity analysis; say $N = 10$ instead of $N = 30$.

Picking hyperparameter sets fairly. We now have a mechanism for measuring idealized performance, but still want to avoid the pitfall where we allow one algorithm to have many more hyperparameter settings than another. We want our results to reflect the utility of our algorithm, rather than reflecting performance when fitting hyperparameters to a set of environments. Unfortunately, there is not an explicit, precise procedure here to obtain a perfectly fair experiment. The only way to avoid tricking ourselves is to attempt to be as fair as possible and to sincerely make choices that are justifiable.

At a minimum, you should ensure all algorithms have the same number of hyperparameter settings that are tested. If one algorithm has two hyperparameters, α and β, and another only has η, then you have to sweep n values for η and n values for the cross-product of (α, β). For example, you might test $\eta \in 2^i$ for $i \in \{-6, -5, -4, -3, -2, -1\}$ and test $\alpha \in 2^i$ for $i \in \{-5, -3, -2\}$ and $\beta \in \{0.1, 0.5\}$, so that they both have 6 hyperparameter settings.

3.3 Evaluating Algorithms for Deployment

This final setting evaluates the performance of a reinforcement learning algorithm under some "realism" constraints, typically modelling a specific deployment scenario. For this setting, we need an *algorithmic* approach to set hyperparameters, or we need hyperparameter-free

algorithms. It is not possible to test many hyperparameters in most deployment scenarios.[11] For example, if a reinforcement learning algorithm is being used to optimize data center cooling, it can be unrealistic to test even a handful of hyperparameters on the system. Instead, we want to deploy a fully-specified algorithm.

One issue in reinforcement learning is that we do not have a general purpose algorithm for hyperparameter selection, unlike supervised learning. In supervised learning, we can use internal cross-validation to select hyperparameters: the best hyperparameters are selected by separating the data into training and validation, and using validation performance as a measure of generalization performance under those hyperparameters. We have no such equivalent procedure for reinforcement learning. We discuss why this is the case in Appendix D.3, and propose potential options for developing such algorithms for reinforcement learning.

A common approach today is to use the default hyperparameters specified in released code-bases. Using these defaults is not unreasonable, if the goal is to compare two systems. In an empirical study, you may want to understand the performance of two code-bases across a variety of different problems. This experiment is not about comparing the algorithms underlying those systems, but rather the systems themselves. It could help a practitioner decide which of the available code-bases might be more suitable for their application. Of course, the default hyperparameters in the code-base were likely set on a small set of simulation environments; so we should be cautious about how well the system will perform in deployment.

Another strategy has been to tune hyperparameters on a subset of environments, and then fix them for a larger set of environments. This practice was used in the Atari suite for example, where it was suggested to use five of the 57 games for hyperparameter tuning (Bellemare et al., 2013).This procedure could mimic a deployment scenario, where you have several simulated environments related to your real-world environment, on which hyperparameters can be tuned. However, as yet there is little understanding of how one might pick such tuning environments.

Key insights: dealing with hyperparamerers

1. Unthoughtful treatment of hyperparameters is one the leading causes of bias in empirical RL. The first step is acknowledging this bias.
2. When designing an experiment, make a plan to select hyperparameters in a way that matches the goals of the experiment, rather than as a frustrating nuisance.
3. Hyperparameter sweeps are not a general-purpose method for finding hyperparameters; instead, we use sweeps in experiments to understand our algorithms.
4. For sensitivity plots, select ranges of hyperparameters mindfully. Ensure the range is wide enough; if you find optimal parameters lie on the boundary, expand the range.

11. The one exception is when the ultimate goal is to solve a simulated environment. This corresponds to only a limited set of environments, and so we do not consider it here. It is important to note that for simulated environments, even though we can use hyperparameter optimization (or sweeps), we then have to account for that as part of the algorithm and as part of the computational cost.

5. Visualizing hyperparameter sensitivity is itself a difficult problem; be innovative about how to communicate this to the reader. Current options include two-dimensional parameter sensitivity curves over one hyperparameter and violin plots of all hyperparameters.

6. In general, it is better to avoid reporting performance for the best hyperparameters chosen in an environment. There are too many pitfalls, and you risk tricking yourself about the quality of your algorithm.

7. Default hyperparameter settings are not necessarily appropriate and certainly not always fair. See more in the next section about the pitfalls of untuned baselines.

4 Comparing the performance of multiple algorithms

Most—if not all—the things we worry about when investigating a single agent are relevant when investigating more than one. However, the concerns become more serious as we are often making a value judgement on the ranking or relations between multiple agents. The claims are inherently more nuanced and thus the standard of evidence and rigour required goes up a notch.

There are many reasons we ultimately compare multiple algorithms. The most common goal is to provide a performance ranking over multiple related algorithms on a given problem setting (or suite of problem settings). Implicitly, we are making the claim: "if your problem is similar to this problem setting, here is the algorithm you should use". Supporting such claims well is rife with difficulties and we will dive into the details in Section 5.2.

An alternative reason to compare algorithms is to show that one algorithm (the baseline) suffers from some problem, but the newly proposed algorithm does not. A prototypical example is the introduction of Gradient TD methods (Sutton et al., 2009). The comparison starts by convincingly illustrating that the baseline algorithm (TD in this example) suffers from some problem by crafting a very specific and simple counterexample. Then we introduce a novel algorithm or modification to the baseline and show that this modification does not suffer the same problem.

In this section, we will explore the challenges that arise for both types of comparison and discuss strategies for drawing reliable and robust conclusions. Naturally, it is impossible to capture all possible future experiments; however, much of the following discussion generalizes well to many possible experimental procedures. We will assume the reader has well understood the preceding sections on summarizing the performance of a single algorithm (Section 2.2) and understanding the impact of hyperparameters on an algorithm (Section 3) as these concerns become even more exaggerated in the multiple algorithm case!

4.1 Designer's curse: with great knowledge comes great responsibility

Evaluating your own algorithm is rife with bias. You know the ins and outs of your algorithm better than anyone on the planet. You have spent months working with different versions of your algorithm, tuning them for performance, finding environments where your algorithm shines and discarding ones where your method does not. The baselines you eventually compare against likely have not received the same attention in the same environments.

Worse, you will not have the same detailed understanding of those baselines, nor knowledge of how to tune them well.

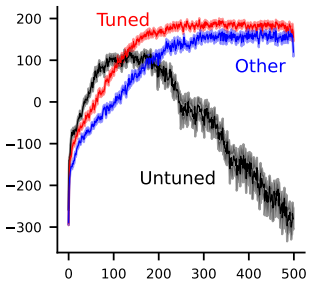
There is nothing we can do about designer bias (in fact it is a nice side effect of good algorithmic research), but there are steps we can take to reduce experimenter bias. However, you can counteract it by spending significant effort to get the baselines working well. It is better to risk giving the baselines a small advantage to mitigate over-claiming.

Let us revisit the example of an experiment where we propose an algorithm to resolve some specific failure case. Implicitly, this experiment is actually making two independent claims. The first claim is that the baseline algorithm experiences that failure case. As we saw in Section 3, it is exceptionally challenging to show that an algorithm has a consistent failure case. We can always ask: *would this happen for a different choice of hyperparameters?*

The second claim is that our proposed algorithm solves the failure case. Illustrating this second case can be much easier; we just need to find a single hyperparameter setting that works! As a result, a truly fair comparison requires spending a significant amount of time understanding the baseline algorithm while requiring far less time (empirically) understanding the proposed algorithm. If we had not spent the time with the baseline, we could likely be building a strawman argument; one which does little good for the scientific understanding of learning algorithms.

When we are benchmarking our algorithm, one of the easiest ways to protect against designer bias is to choose the right environments. We can let the authors of the baseline algorithm do the work here. If you are comparing your new algorithm to *Soft Actor-Critic* (SAC) (Haarnoja et al., 2018), then use an environment where prior work has shown SAC does well. It is reasonable to assume that the authors of SAC worked hard to tune their own algorithm—that is designer bias working for you! It is important that you pair the environment choice with fully specified algorithms because our algorithms are not yet general. SAC with hyperparameters tuned for HalfCheetah, for example, is likely to not work well in Mountain Car. It is okay to try a baseline algorithm on a new environment, but then you will likely have to expend significant energy tuning it (perhaps changing the network architecture, optimizer, etc.) and you really cannot be sure you will do a good job.

Example: Untuned baseline.



The inset figure shows one example where an untuned agent is applied to a new environment, in this case Lunar Lander. Imagine we have introduced a new algorithm called DeepQ—it is a simple action value learning agent with a Q-learning update and neural network function approximation (the blue line). Now imagine we grabbed a freely available DQN implementation with a default set of hyperparameters tuned for some other problem. Comparing the blue and black lines might lead one to think DeepQ is much better than DQN on this task, but we are misleading ourselves. The red line represents DQN after tuning its hyperparameters for Lunar Lander. *Voila!* Much improved performance and ego checked! The moral of the story is simple: **beware of untuned baselines!**

4.2 The utility of calibration baselines: what does that line even mean anyway?

Sometimes comparing the performance of two algorithms is not enough because we lack context to understand the results. Returning to the Lundar Lander experiment above we might naturally wonder: are any of these algorithms doing well? How would a non-learning policy like random action-selection fair? What about a learning agent with a much simpler function approximator like SARSA(λ) with tile coding? Anecdotally, years ago when one of the authors submitted their first paper on using reinforcement learning to play hearts, pesky Reviewer #2 said roughly: "sure your agent is good compared to human players and the best search agent, but I don't have any basis to understand the performance. You should include results playing against a random agent". Reviewer #2 was right. We included the baseline, which ultimately showed our agent and the other baselines were very strong indeed, and it made our paper better. You should always ask yourself if such baselines—not just SOTA algorithms—could make your results easier for the non-expert to interpret.

Picking the right calibration baseline depends on the research question, but generally there is a lot of flexibility here. It is often good to think of both randomized and high-performance or even oracle baselines. Oracle baselines often have access to side information that your learning agent does not, setting an unobtainable but interesting performance bar. Alternatively, an oracle baseline might just operate under different constraints than your new method. For example, we often compare linear complexity off-policy TD methods against least squares TD (LSTD). LSTD is a quadratic method that performs more compute per step than a linear method and thus naturally we expect it to set a high performance bar. If your new linear off-policy TD method approaches the performance of LSTD on several environments, you can be more confident your method is learning efficiently and that your method's improvement over other linear baselines is relevant.

While these calibrations provide useful context for interpreting results, we must be cautious not to become over-reliant. For example, the Atari suite uses random performance and human-level performance as calibration baselines (Mnih et al., 2013). However, humans have rather different constraints than our agents; requiring a screen to project light to our eyes, having latency between our brains and fingers, the often imperfect translation from muscle activation to controller input, and finally latency from controller to gaming system. Achieving human-level performance is certainly informative of learning capacity and progress; exceeding human-level performance by many orders of magnitude, however, may be a result of fewer constraints. As we move increasingly far from our calibration baselines, we must re-evaluate their utility.

4.3 Ranking algorithms (maybe don't)

A common goal is to show that your new algorithm is better than some prior work. However, providing supporting evidence towards this claim can be exceptionally difficult. In fact, in all but the most rare cases, we might go so far as to say that providing such evidence is entirely infeasible! The space of problem settings where any given algorithm may be deployed is large and wholly unspecified; showing that any algorithm generally outperforms another across this space would be impossible without first defining the space. Even once this massive problem-space is well defined, gathering sufficient evidence to cover the space is likely intractable.

Instead, we tend to produce a ranking of algorithms on a small problem set and hope our claims generalize. Let us discuss a bit more why this might be problematic. Even if we could show improved performance for a particular benchmark, this is likely not broadly informative. It is like trying to generalize from a biased distribution (say performance in Atari) with a few samples to make predictions about performance across the whole set (performance in all environments).

Instead, we argue that insights about *why* an algorithm behaves differently tend to generalize. Highlighting issues using synthetic environments, like counterexample MDPs or environments with carefully crafted properties such as Riverswim (Strehl and Littman, 2008), allow us to reason about what situations *will* cause an algorithm to fail. It is up to domain experts, then, to identify if these failure modes apply to their problem setting.

Example: The deadly triad.
One such insight that has aged well is the *deadly triad* (Sutton and Barto, 2018). The deadly triad specifies three conditions where TD-like learning algorithms will often fail: (1) when the algorithm learns from bootstrapped estimates, (2) off-policy samples, and (3) function approximation. Given this set of conditions, then, it is not hard to propose an environment where TD algorithms fail; many have been proposed in the literature (Baird, 1995; Kolter, 2011; Tsitsiklis and Van Roy, 1997). These conditions have been further refined over the years to include properties of the environment (Kolter, 2011; Patterson et al., 2022b) and properties of the function approximator (Ghosh and Bellemare, 2020) allowing an even more crisp understanding of when algorithms might fail. These insights have later been shown to be robust across many different learning problems, architectures, and fully engineered learning systems (van Hasselt et al., 2018; Obando-Ceron and Castro, 2021; Patterson et al., 2022b).

Another challenge that arises is that of fairness. When comparing the performance of multiple algorithms, many design choices must be made: setting hyperparameters, picking which environments to use, how long to run each experiment, and so on. Sometimes these choices have non-obvious and indirect impacts on the performance of each tested algorithm, often leading to latent unfairness in the experimental design. A noteworthy example is the experiment length used for the Atari benchmark. Shortening the length of experiments run on Atari dramatically changes the ranking of algorithms (Machado et al., 2018; Agarwal et al., 2021), leading one to wonder which ranking to consider when applying any of these algorithms to your problem of interest.

Sometimes it is not obvious how to maintain fairness across different algorithms. For instance, if two algorithms have wildly different architectures, it can be challenging to ensure representation capacity is fair. Instead, an empiricist might fix the representation capacity of one algorithm, while testing the other algorithms with increasingly large representations. All levels of the varied axis are reported giving a complete picture of the performance difference as representation capacity changes.

Unfortunately, it can be expensive to exhaustively test an algorithm for multiple settings of a confounding variable—in fact, this is effectively a form of sensitivity analysis akin to Section 3. When designing fair comparisons, we often come back to focus on understanding our algorithms, rather than ranking them. We discuss this further in Section 5, where we discuss selecting and designing environments for experiments.

If this cost is prohibitive, an alternative option would be to explicitly run an unfair experiment, providing a handicap to our own proposal algorithm. Often, our proposed algorithm already has multiple sources of implicit unfairness: we spend more time on the implementation, we better understand the algorithm and when it may fail or succeed, we spend more time tuning the algorithm to our problem setting. When we provide an explicit source of unfairness—such as a reduced representation capacity—and our algorithm still outperforms competitors, we show a lower bound on the potential improvement provided by our algorithm, implying that an even greater degree of improvement would be observed under even more fair conditions. Naturally, we should be careful not to overclaim here; we cannot know to what extent our algorithm improves performance under more fair conditions.

4.4 Statistically significant comparisons for two fully-specified algorithms

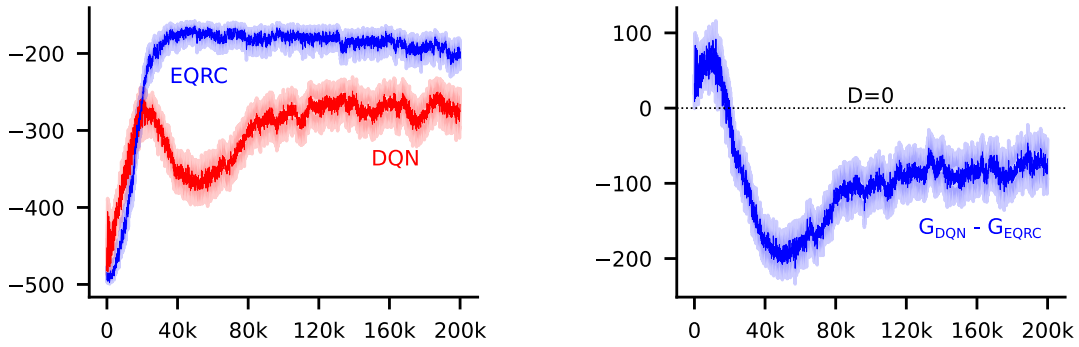
Imagine that we want to compare Algorithm A and Algorithm B, in Mountain Car, in terms of the online episodic return, in expectation across many runs. We may want to say that A is better than B, with high confidence, on Mountain Car.[12] One of the simplest strategies for comparing the two is to compute confidence intervals, which we discussed for a fully-specified algorithm in Section 2.5. If the two intervals do not overlap, and the mean value for A is above B, then we can conclude that A is statistically significantly better than B.

However, this is a low powered test. In other words, depending on the number of runs, the confidence intervals may overlap but a more powerful test, like the paired t-test, might have allowed us to conclude that A is statistically significantly better than B. The primary reason is that pairing allows us to account for sources of variation due to the environment or initialization. For example, for one random seed, the agent may be started in a difficult start state, impacting all future learning. If we compare the two agents for that seed, then we may find that both performed poorly for that seed, but that the relative ranking remained the same. The small modification to get the paired t-test is simply to look at *differences* in performance, rather than the performance itself.

We can leverage the same idea to visualize confidence intervals for learning curves. Overloading terminology, let the performance of one algorithm be random variable A and the performance of the baseline be random variable B. Then we would report an interval around $D = A - B$ as opposed to reporting two intervals (l_A, u_A) and (l_B, u_B), as shown in Figure 12. Whenever the lower bound of the interval is greater than zero, our proposed algorithm outperforms the baseline with our stated level of confidence. Notably, this has the additional advantage of removing one line and shaded region in learning curve plots—at the cost of a slightly obfuscated visualization of performance. In the end, both plots may be desirable.

It may actually be preferable to use this confidence interval on the difference rather than a hypothesis test because it provides more information. In particular, it highlights the magnitude of the difference between algorithms, called the *effect size*, not just that they are different. Most hypothesis tests are designed to answer the question: is the average performance of algorithm A better than the average performance of algorithm B? If there truly is a difference in performance between these algorithms, then with enough samples we will eventually be able to reject the null hypothesis and provide evidence that A is better

12. Presumably we are verifying that A resolves an issue with B, that manifests on Mountain Car, rather than attempting to outperform B on Mountain Car.



(a) Individual performance of DQN and EQRC. (b) Performance difference $D = G_{\text{DQN}} - G_{\text{EQRC}}$.

Figure 12: In **(a)**, we see the performance of two algorithms, DQN and EQRC (see (Ghiassian et al., 2020)), on the Mountain Car environment. Shaded regions show individual confidence intervals for each algorithm. In **(b)**, we show the performance difference $D = G_{\text{DQN}} - G_{\text{EQRC}}$. When $D > 0$, then DQN has better performance. When $D < 0$, then EQRC has better performance. And when the shaded region **does not** include the horizontal line at $D = 0$, then the reported difference is statistically significant. It is challenging to identify if there are differences between algorithms in the early portion of learning in the individual curves in (a), but it is much clearer to see in the difference curve in (b).

than B at the $p = 0.05$ level. However this ignores a very important nuance: the difference in performance may be negligibly small. A common solution to this problem is to include an appropriate measure of the effect size for the chosen hypothesis test, for instance for a t-test one might include Cohen's d to measure the effect size. However, such effect size tests may not be available in all situations and can often be difficult to interpret.

Another common criticism of hypothesis tests are that their results are easily misinterpreted; in fact, this has lead to some scientific publications banning the use of p-values (Wasserstein and Lazar, 2016). A common misinterpretation of the p-value is that p represents the error rate of the test. Rather, the p value says: *if the null hypothesis were true, then the probability of seeing results at least this extreme is p due to sampling variation.* However, naturally, we do not know whether the null hypothesis is true—this is why we are performing the test—and so we cannot know the true error rate from this alone. Recent statistical simulations suggest that, for sensible assumptions, the true error rate for a p-value of $p = 0.05$ may be as high as 26% (Sellke et al., 2001; Colquhoun, 2017). As the p-value decreases and the effect size increases, this error rate quickly drops.

The principle behind pairwise comparisons is also about controlling randomness whenever we can. The ability to replicate and reproduce results is core to scientific investigation. Computer simulations have a particular advantage of being nearly perfectly replicable. To do so, however, we must carefully control sources of variation: this means controlling the pseudorandom number generators responsible for simulating randomness in our environments and agents. A common approach is to set a global random seed for the entire simulation, impacting the random state of all randomized components simultaneously. However, it can often be beneficial to retain independent random states for each randomized component of

the experiment. We give an example below about how this could be useful when evaluating the addition of auxiliary losses, measuring a property called the *stable rank*.[13]

Example: Separating the random seed for the agent and environment.
Imagine you want to understand the impact of adding an auxiliary loss to your agent. Specifically, your hypothesis is that the stable rank of the agent's representation will decay less quickly when you use an additional neural network head to predict the next state. To test this hypothesis you start with a single environment—say Puddle World—select a learning algorithm with all hyperparameters specified except for the neural network architecture, then create two fully-specified algorithms by defining a neural network architecture with the auxiliary head, and another without. Finally, you run each algorithm 30 times, measuring the stable rank of the neural network layers periodically through the run for each individual agent.

Consider the code-design where you set a single global random seed for the entire simulation and you run the first agent seed = 0 for both architectures. Because both neural networks have different sizes and both are initialized randomly, each agent will call the random number generate a different number of times—at the end of agent initialization, both random number generators are in different states. Then your code initializes the Puddle World state by randomly selecting a starting coordinate. However, because both RNGs are in different states, both investigated agents likewise start in different states. Because the starting state in Puddle World considerably changes the sequence of observed rewards, the agents for seed = 0 receive wildly different data.

On average over several different agents for each experimental condition, the effects of differing observation sequences will wash out and we will be able to detect if our auxiliary loss plays a role on stable rank. However, if we had used individual random states for both the agent initialization and the environment initialization, the two seed = 0 agents would have observed the exact same sequence of observations and would share some degree of variation due to these observations. Taking advantage of this pairing structure, we can detect differences in stable rank with far fewer agents by cancelling out nuisance sources of variation. This experimental design is called **repeated measures**.

4.5 Statistically significant comparisons for multiple fully-specified algorithms

There are times we want to compare multiple agents. For example, there may be a baseline, an existing algorithm and your modification of that algorithm. A sensible choice is still to choose an algorithm as a comparator, and plot differences to that algorithm for the rest. This approach mitigates some of the variance we might see due to the particular configurations in a run. We can report confidence intervals for these difference curves, as above.

However, we need to consider issues with *multiple comparisons*. Imagine that we compare 5 different algorithms on a plot, all with performance differences to some baseline. We may mentally compare the 5 different confidence intervals, or run pairwise comparisons using hypothesis tests. If we treat each comparison independently, then the failure probability δ accumulates. To account for these multiple comparisons, the simplest solution is the

13. The stable rank is the ratio between the Frobenius norm and the spectral norm of the weights. It was introduced as a measurement of capacity, for instance Arora et al. (2018).

Bonferroni correction, that uses $\delta/5$ to compute these accumulating probabilities to allow for a confidence of $1-\delta$. Naturally, this requires even more runs (and more compute) to ensure statistical significance.

One of the simplest solutions to this issue is to narrow the scope of our experiment, and run fewer algorithms. We can limit the scope of baselines to those that have similar design philosophies or characteristics to our own proposed algorithm. For example, perhaps we propose a model-free policy-gradient method and so choose to only compare to other model-free policy-gradient baselines. This restriction narrows the question to algorithmic improvements within the same class of algorithms, rather than also trying to show improvements for policy gradient methods over an entirely alternative class like model-based Q-learning algorithms. It is actually reasonable to separate this for clarity reasons, not just computational reasons.

If, on the other hand, your empirical goal is to compare a large set of algorithms, then the answer is simply that you need to do more runs. It can be useful to investigate if policy gradient methods have advantages over an alternative class like model-based Q-learning algorithms. When answering such a broad question, naturally it requires more compute. This further highlights why it is useful to separate these concerns. It is worthwhile to have empirical studies and benchmark papers that attempt to run these more comprehensive experiments. This is a mountain of work! It is infeasible to also include such work within a paper focused on a new algorithm.[14]

Another common approach is to isolate the precise modification made in the proposed algorithm. Perhaps the proposed algorithm introduces a new sampling strategy from replay buffers. Instead of comparing against a suite of different learning rules, we can instead make a small number of pairwise comparison between similar algorithms. For instance, we can endow DQN and A3C with our proposed replay buffer and perform two pairwise comparisons against the original DQN and A3C respectively. We do not care if DQN with the novel replay buffer outperforms A3C with the old replay buffer, as many design variables change between these populations. By eliminating several such pairwise comparisons, we can significantly decrease the probability of false negatives and number of samples to detect differences.

Key insights: comparing the performance of multiple algorithms

1. Be aware of your designer bias. Ultimately, you want to avoid misleading yourself, as well as the rest of the community.
2. Consider the use of baselines (e.g., random, oracle) to contextualize performance.
3. The purpose of comparing to an algorithm is primarily to show that your new approach resolves the issues that you claim it does. It is difficult to claim that you have a generally better algorithm only using performance across several environments. Design experiments to provide insights rather than state-of-the-art claims.
4. We can leverage paired comparisons between algorithms, to reduce variability and make statistically sound conclusions with fewer runs.

14. It behooves us to recognize that this is precisely what Reviewer 2 often asks for: benchmarking. The only answer, for scientific rigor, is to resist the siren call.

5. Statistical intervals—confidence intervals and tolerance intervals—provide a more interpretable alternative to hypothesis tests, when comparing two algorithms. By reporting intervals, we provide both a sense of effect size and an estimated error rate.

6. Consider setting the random seed for the environment and agent separately. This *repeated measures* empirical design allows us to control for stochasticity in the agent (e.g., neural network initialization) separately from environment stochasticity (e.g., start state), ensuring pairwise comparisons share more similarities. Remember, the seed is NOT a tuneable hyperparameter.

7. Comparing more than two algorithms requires more care, and we can run into *multiple comparisons* issues. Err on the side of designing experiments where a more focused questions is asked, comparing a smaller number of algorithms.

5 Selecting environments

The selection of environments is a critical part of the setup of your experiment. In other sciences, experiments are about the natural world and are naturally constrained. The natural world provides a rich suite of problems for us as well, such as robotics, chat bots, controlling power plants, trading, and more. These real-world problem are often where we want to eventually deploy our algorithms; simulation is where we can prototype. In simulation, it is difficult to maintain the richness of the real-world. And worse, we may inadvertently design environments that are actually impossible or too easy.

When developing an algorithm, the first step is usually to hypothesize a very simple environment and experiment where you are highly confident in the outcome. For example, to test an agent's ability to remember, we might design a small gridworld where the agent must remember that pressing a button unlocks a reward. This first step provides a foundation for more complex experiments. If there are surprising results in this first step—as there very often are—then it can be much more carefully understood before moving to a setting where it can be harder to isolate the issue. It is also a step where a surprising result can make you rethink the algorithm itself, thus providing conceptual clarity. We discuss such *diagnostic* environments in Section 5.1.

After that, the next step is to evaluate the algorithm(s) on existing benchmark environments. We discuss the role of using benchmark environments further in Section 5.2. We conclude the section with a brief discussion about difficulties with creating new environments and about the potential utility of aggregating environments.

5.1 Diagnostic environments

The first step in designing a diagnostic MDP is to isolate the key issue your algorithm is designed to solving. If you can identify that issue, then you can design an environment that is defined by that issue or where the issue is exaggerated enough to find an effect—that is exactly what a *diagnostic MDP* is. In this section, we will give several examples of effective diagnostic environments and explain how they provide this exaggerated effect.

A classical diagnostic environment is Baird's counterexample (Baird, 1995). This 7-state environment was designed to show that temporal difference (TD) learning divergences

under off-policy sampling. This diagnostic environment may seem like a counterexample, rather than an environment. However, it is an ideal example of a diagnostic environment because it is precisely one where the experimenter should be quite sure of the outcome—divergence—but cannot perfectly anticipate actual behavior in deployment. Surprising outcomes have since come out of this environment. In our own experiments, for example, we have found that incorporating certain adaptive stepsize algorithms into TD seems to resolve this counterexample, though there is no theory that suggests it should. Further, with a λ larger than $1/7$, the GTD(λ) algorithm exhibits instability in this environment (White and White, 2016), providing useful insights into the role of eligibility traces in off-policy learning.

Such diagnostic MDPs themselves can have lasting impact on algorithm development. Baird's little MDP has been repeatedly used to test new gradient algorithms. In some cases, it has highlighted issues with those algorithms. Emphatic TD, for example, should converge on Baird's counterexample in theory, but in practice the variance is sufficiently high that convergence is very poor and requires careful tuning (Sutton et al., 2016; Mahmood, 2017).

Diagnostic environments are chosen based on the hypothesis you wish to test, which means that existing diagnostic environment may already have been designed to isolate that property that you wish to test. For example, you might hypothesize that an algorithms is prone to settling on a suboptimal policy because it does not explore enough. You might select the classic environment Riverswim (Szita and Lorincz, 2008), which was designed specifically to test this. The environment requires the agent to move to the right—upstream—to get to the highest reward in the rightmost state, but the dynamics push it to the left—downstream. In the leftmost state there is a misleading positive reward that further encourages the agent to settle on this easy, but suboptimal, solution.

Sometimes you may need to design your own diagnostic MDP. How might one go about doing so? As a personal example, we needed an MDP to highlight when a standard off-policy policy gradient method, called OffPAC, might diverge (Imani et al., 2018). The purpose was to highlight a soundness issue in OffPAC, and why the state-reweighting proposed in the new algorithm resolved this issue. Designing this diagnostic MDP required thinking about how to make the solution under OffPAC select the wrong action, due to giving too much weight to an unimportant state. The idea was to alias together two states with different optimal actions, forcing any learned policy to have to select the same action in these two states. OffPAC incorrectly gave the state along the suboptimal trajectory higher weight, whereas the new algorithm reweighted the states to give higher weight to the state along the optimal trajectory. The procedure towards designing this diagnostic involved understanding the source of the issue, and considering a minimal setting where it might arise.

5.2 Benchmark environments and challenge problems

Benchmark environments provide a useful tool to assess if there are issues with a new algorithm—a sanity check. The word *benchmark* means a point of reference. The possible performance on these environments is well understood, due to previous results, making it easier to understand if you are doing better or worse than this reference. If you are doing notably worse than a known reasonable solution, then this could indicate an issue with your algorithm that should be addressed.

For this reason, benchmark environments can remain useful for many years. Classic environments that are still commonly used include Mountain Car (Moore, 1990), Cartpole (Sutton and Barto, 2018), Puddle World (Sutton and Barto, 2018) and Acrobot (Sutton, 1996). These simple environments play a useful role because we understand very well how to get good performance in these environments, and so we can easily see when there are issues in new algorithms. For example, a SARSA(λ) agent with tile-coding can find a good policy in Mountain Car in a dozen or so episodes. DQN—using replay and target networks—learns much more slowly, settles on a worse policy, exhibits more instability, and is sensitive to the target network refresh rate (Hernandez-Garcia and Sutton, 2019; Kim et al., 2019; Patterson et al., 2022a). Running DQN on this environment helps identify that there may be room for algorithmic improvement.

Certain environments may start as challenge problems—ones that we do not know how to solve well—and eventually become benchmark environments. Two examples of such benchmarks are the Atari suite (Bellemare et al., 2013; Machado et al., 2018) and Mujoco environments (Todorov et al., 2012). A challenge problem plays a different role than a benchmark environment. It highlights gaps in all of our methods; trying to fill these gaps can drive algorithm development. If an environment is truly a challenge problem, then experiments are more *exploratory* or *demonstrative*. It can be sufficient to show that you can obtain good performance—demonstrate something is possible—without even comparing to any other methods or only including basic baselines. The results suggest that you can do something that was not possible before.[15]

As our methods improve on these challenge problems, they become benchmark environments. At some point, however, they can be in a confusing interim stage. We as yet do not have an understanding of how to obtain good performance, but methods have gone from being abysmal to merely mediocre.

It is also at this stage that we start to overfit to these interim benchmark-challenge problem environments. The community tends uses these benchmark-challenge problems to gatekeep new work: requiring new work to show clear improvements in these environments, with comparisons to the now growing list of algorithms that perform okay in these environments. There is a tendency to dismiss any experiments in simpler benchmark environments, since those environments are "too easy". Eventually, this causes overfitting to these environments to eke out ever smaller wins. It is an issue that happens in reinforcement learning, as well as on benchmark datasets in machine learning.[16]

This direction is typically not beneficial for general algorithm development. Instead, it may be better to acknowledge this transition and begin using these environments as

15. This is not how Atari is now used in the RL literature, but the first experiments including the original DQN paper (Mnih et al., 2013) were very much of this form.

16. And machine learning is, of course, not the first place to experience these pains. Benchmarking also became a serious issue in planning and search (Hooker, 1995). A particularly evocative quote is as follows. "It would be absurd to ground structural engineering, for instance, solely on a series of competitions in which, say, entire bridges are built, each incorporating everything the designer knows about how to obtain the strongest bridge for the least cost. This would allow for only a few experiments a year, and it would be hard to extract useful knowledge from the experiments. But this is not unlike the current situation in algorithmic experimentation. Structural engineers must rely at least partly on knowledge that is obtained in controlled laboratory experiments (regarding properties of materials and so on), and it is no different with software engineers." (Hooker, 1995)

benchmarks sooner. This means it is not key to outperform whatever (overfit) solution is considered the current best, but rather to take a useful baseline with well-known performance as a sanity check on your algorithm.

For all the above reasons, it is important to remember that most of our experiments in benchmark environments are to identify issues with our algorithms, rather than to make bold claims about state-of-the-art performance. It is clearly useful to identify and fix issues in our algorithms using experiments. It is not as clearly useful to rank algorithms based on performance in small simulation environments. If there are stark and meaningful differences in well-known environments, then that success can and should be reported, with hopefully accompanying experiments to understand why. Inability to get stark improvements on benchmark problems, however, should not prevent pursuing an idea, nor gatekeeping others in pursuing ideas. Your experiments should highlight one setting where your new algorithm is demonstrably useful with a clear explanation of why. You can also demonstrate acceptable performance on benchmark environments, to show nothing is obviously broken.

5.3 New (or modified) environments are not always better

Designing environments is hard. It is also arguably something many of us have little expertise in. We learn a lot about developing algorithms, much less about developing environments. It is important to realize there are relatively few well-known environments that have stood the test of time, and the ones that do have been refined and fined tuned over years, if not decades. For example, Mountain Car was first proposed as the Puck on a Hill problem by Andrew Moore, in his PhD work (Moore, 1990). The problem featured continuous actions and a non-zero reward for reaching the goal. Years later, Sutton and Barto changed the dynamics representing the hill as a cosine wave, made the actions discrete, and the reward -1 per step. This later version became the standard for over 20 years, before AIGym introduced an aggressive episode cutoff of 200 steps to improve the performance of neural network learners.

Take pause when deciding to invent a new environment and consider the costs. First you need to justify clearly why we need yet another environment. The biggest concern is that you may very well invent an environment that is invalid or needs further improvement. But practically speaking, you are making more work for yourself. If you use your new environment to highlight the merits of your new algorithm, then you must retune baseline methods to avoid reporting results with untuned baselines, as we have already discussed in Section 4.1. This can be a lot of work and is error prone. Exercise similar caution when modifying existing environments. These changes need to be justified, particularly as it means that older benchmark performance no longer applies, losing one of the key reasons to use a benchmark environment in the first place. Sometimes the easy road and the more scholarly choice is to use an existing environment.

These changes, or new environments, also might cause us to accidentally design environments in support of our solution method. For example, we might run DQN on discrete-action Pendulum swing up, and find under standard configurations it performs surprisingly poorly. We then might try a few changes, to maintain most of the essence of the environment but make learning more feasible. One such change could be introducing episode cutoffs and random start states to facilitate exploration. Now we have a new environment that has potentially been tuned to be easier for methods like DQN. If we test a policy gradient method,

or a completely new approach, then we may have inadvertently favored DQN. **Co-evolving our problems and solution methods in this way is always dangerous.** Note, this is not the same as designing new diagnostic MDPs. A diagnostic MDP allows us to highlight the key issue being tackled. It is common for papers to introduce new diagnostic MDPs—that may never be used again—to make a clear conceptual point.

One route for environment creation is to consider *environment collections*. For example, we may collect several classic environments like Mountain Car, Cartpole and Acrobot into a Classic Control set composed of problems with non-imaged based, low-dimensional observations. Or we may collect several Atari environments that seem to be more difficult in terms of exploration into an Atari Exploration set.[17] We can then report performance in aggregate, across the set, as one macro-environment, without explicitly considering performance on each environment in the set.

There are several benefits to this approach. This design tests the performance of the method on a set of related environments with a particular property, making conclusions less specialized to one specific environment. Mainly, it shows that some emergent phenomenon from an algorithm (e.g., decorrelated features) arises across several environments. Of course, we still need to understand *why* our method behaves the way it does. Using more environments does not allow us to conclude our method will perform better, but it does at least show that the emergent phenomena arises in more than one environment. Additionally, we obtain this ability to test across environments with minimal increases in compute over testing in one environment. That is because we can do fewer runs within each environment, to still get a large number of runs in the macro-environment (collection). A related strategy, to test generalization, was proposed earlier by having systematic parameters that could be varied for an environment (Whiteson et al., 2009), such as gravity in Mountain Car.

Remark 4

Considering macro-environments has some benefits, but there are also challenges in aggregating performance across environments (e.g., dealing with different reward scales). We discuss this topic in more depth in Appendix E.

Key insights: environment selection

1. Small diagnostic environments can be used to highlight the properties of a method—called issue-oriented research. These diagnostic environments can provide conceptual clarity as well as a critical sanity check.
2. Designing environments is hard; do not assume an existing environment is reasonable.
3. Further, ramifications on agent behavior can be large from even minor changes to the environment—such as adding random starts or adding episode cutoffs. Proceed

17. Even when just aggregating environments, we need to be careful that we have designed the collection to do what we think it does. For example, Taïga et al. (2019) showed how prior work focusing on so called hard exploration domains in Atari caused researchers to miss that simple ϵ-greedy methods were actually better than count-based methods across the whole Atari suite. Environment design is hard.

with caution when changing an existing environment and make sure you have a good justification for not using the original standard.

4. It is not better to run on more environments than less environments. It depends on the empirical question being asked. If you find yourself thinking you should run on one more environment, then make sure you ask yourself why and select the next environment deliberately rather than to simply increase the number of environments.

5. It is unlikely your algorithm will perform best on all environments. But, it should have the advantages you claim it has. Consider making a macro-environment (group of environments) for which your algorithm should be better, and another macro-environment where it may be worse. Verify if your hypothesis is correct, and report both this good and bad performance.

6. Grouping environments adds structure to an otherwise mis-mash of results showing good and bad performance. However, do not post-hoc group environments based on performance. Groupings should be performed beforehand based on some criteria, to test a hypothesis. Changing the hypothesis after the fact is called *harking* and results in instant shame, especially from psychologists.

7. It is not acceptable to run an incomplete experiment due to insufficient resources or compute, even if you think the reviewers will not accept your paper. Design and run a meaningful experiment in an environment that is feasible for the resources you have available. You will simply have to be more clever than those with bountiful resources.

In the end we generalize our conclusions from the small set of experiments we conducted. We can only make limited claims about the generality of the method, even if we run on many environments. In fact, we can only make more measured claims about the outcomes we observe, under the particular conditions in which we test the methods. This includes the particular set of environments, and the hyperparameter selection strategy, and how many steps of interaction were used, and so on. If designed well, however, such measured claims can be highly meaningful: the key outcome is improved understanding of the properties of an algorithm. Insight-driven outcomes should generalize. These insights help direct further algorithm development and experiments. This contrasts with demonstrations where the goal is to show a proof of existence, without necessarily understanding the reasons explaining the observed performance—it is harder to generalize these outcomes. This is one of the reasons it is so critical to specify a hypothesis and carefully test it: to make our experiments more meaningful.

6 Common errors in RL experiments

We conclude this paper by outlining a list of common errors in reinforcement learning experiments. Up to now, this document has focused largely on best practices. Naturally, it is jarring to focus too much on all the wrong choices that could be made. But both positive and negative examples of empirical design are needed to become the best empirical practitioners. Nothing in life is black and white; what we identify below are things that generally should

be avoided, but sometimes they might be appropriate. Finally, this is meant to be an ever evolving list that we will update continuously.

Averaging over 3 or 5 runs Sometimes it is ok to use as few as three runs. For example, if the agent and environment are both deterministic. In the vast majority of cases this is a risky practice. Even if we compute the variance across runs we cannot be sure we are not simply under-estimating the variance due to luck.

Using others agent code (including hypers) as is Our algorithms and their implementations are not yet general. We are often tempted to simply download some code and run it to get a baseline for our plots. People often defend this practice claiming its ***fair***, but it is anything but! This only makes sense if (1) the implementation is trusted, and (2) you know the performance is representative of the baseline algorithm. Both these things are likely true if the code is from the original paper (introducing the algorithm you are using as a baseline) and the environment you are using is the same as the original paper. Otherwise proceed with caution.

Untuned agents in ablations We may take an existing agent and ablate components. The agent was potentially tuned *with* those components, and there is no reason to believe the same hyperparameters will be reasonable for the agent with components removed or added.

Not controlling seeds In simulation, we get to control all sources of randomness—we should take advantage of that! When we compare two experimental conditions (for instance comparing two algorithms), we should try to minimize the number of differences between the conditions: make sure random weight initializations are the same, make sure initial state in the environment is the same, make sure the same indices are sampled for the replay buffer, etc. These sources of randomness can amount to a huge amount of variance, but if both experimental conditions share some of the same variations then fancy analysis techniques can leverage this joint variance allowing for statistically significant claims with fewer runs.

Cutting off episodes early Terminating episodes early can have a large impact on exploration (making the problem easier) and likely the distribution of performance. If cutoffs are used, then they should be set to be relatively large. For example, in Mountain Car, an aggressive cutoff is 200 steps; a much more reasonable option is at least 10,000 steps.

Treating episode cutoffs as termination Setting $\gamma = 0$ and making an update when the episode is cutoff is incorrect. Consider a cost to goal problem with reward of -1 per step. Termination is good and states close to the terminal state have higher (less negative values). Updating states where episodes are cutoff—which may be nowhere near the goal state—incorrectly increases the value of those states. Instead, either this last transition should be discarded or the agent should bootstrap off the final state before the cutoff. See Section 2.7.

Randomizing start states The start states in an episodic MDP are part of the problem definition. In many cases they are chosen for very particular regions: e.g., the bottom of the hill in Mountain Car. Exercise extreme caution when changing a problem specification—especially if your motivation is to make it easier for your agent.

HARK'ing (hypothesis after results known) It is best to specify the setup of your experiment ahead of time and make a hypothesis. It is important to specify the expected outcome (e.g., my new method will decrease the number of steps until the first positive reward is observed). This is the scientific method.

Do not do the following. Set up an experiment. Make no hypothesis. Run the experiment and look for a performance measure where your algorithm looks best compared to the others, continually re-running the experiment (possibly on different environments each time) until your algorithm wins.

Environment overfitting A pitfall is to produce agents overly specialized to a small number of environments. For example, you might take DQN and Pong and continually add new components and hyperparameters until you improve on DQN. This is unlikely to yield generally useful algorithmic innovations and little to no insight is generated. Another example is the focus on hard exploration games in Atari as extensively investigated by Taïga et al. (2019).

Overly complex agents Generally speaking we are interested in the simplest agent that works well, with the fewest components. The alternative—a more complex agent with similar performance—by construction must have components that contribute nothing. The contribution is figuring out which components matter and why. The instinct when a method is not working is to *add* rather than *subtract*. By focusing on *why* it is not working, rather than how to fix it, you are less likely to fall into this pitfall.

Choosing γ incorrectly One example is choosing $\gamma = 1$ for an episodic problem with only a non-zero reward at the goal. The agent has no incentive to terminate: an episode of length 10 or 10,000 have the same return. For cost-to-goal problems, with -1 per step, it is most sensible to set $\gamma = 1$, and that it what we likely report: number of steps to goal (not discounted steps to goal). The agent itself might still use a γ for learning purposes.

Reporting offline performance while making conclusions about online agents Offline performance typically means the agent is subjected to periodic test episodes where the agent is teleported to a new state, learning is paused and we measure the cumulative reward achieved. Online performance is simply cumulative reward achieved during regular operation—during exploration and learning. Under offline performance there is no penalty for exploration during learning, whereas in online learning there is and thus the agent must trade-off exploration and exploitation. Machado et al. (2018) discusses this at length.

Invalid errorbars or shaded regions First, whatever method you used to construct intervals, you should understand what assumptions on the underlying data are being made. Second, do not just put error bars around anything. We have already shown how our error bars can be misleading. A first warning sign to look out for is if your shaded regions are tighter than the variation in the mean across the learning curve.

Using random problems They tend to not look much like the problems we care about (e.g., random MDPs). Use with caution.

Not reporting implementation details One should be able to use your pseudo-code to implement your algorithm. Special tricks used to improve performance are part of the algorithm and should be reported clearly.

Not comparing against stupid methods Can a much simpler method or even a random policy solve your problem? Does the a simpler method outperform your method? It does not matter if prior SOTA methods did not use these baselines, you should!

Bugs Never trust your code or your results. Finding bugs is good; it means one less bug.

Running inefficient code This error might not seem pertinent to our experiments. However, if you run inefficient code, it hinders your ability to run careful systematic experiments. It inadvertently causes you to change the question you want to answer. An important first step when running experiments is to ensure you have optimized your code, so that you have more flexibility to do more exploratory experiments and feasibly run the final experiments for your paper. Always profile your code.

Attempting to run an experiment beyond your computational budget It is tempting to say it is not possible to do sufficient runs, or not possible to carefully test hyperparameters, due to lack of compute. So you conclude it is only feasible to do a smaller number of runs. *However, this is not a valid claim.* Let us return to our example of animal learning researchers and rats. If a lab can only afford one rat, then it is not acceptable to run experiments using only this one rat. Instead, they simply have to do different research. The same is true for us. If you do not have the resources to run a correct experiment (say in Atari), then the alternative is not to run an incorrect one. Instead, the alternative is to run a different experiment entirely, one that is feasible and meaningful.

Gatekeeping with benchmark problems This is more about reviewing than running good experiments, however, there is such a strong connection between the two we would be remiss not to discuss the perils of benchmarking. Experimental results should be evaluated in how they contribute understanding and insight. Algorithms tuned to benchmarks often have many small tricks. It is often unclear which tricks contribute to success. Demonstration results *can* be helpful—especially to inspire progress—but not all (or even most) experiments should be demonstration results.

7 Case study: re-evaluating previous work

Congratulations! You made it all the way through and have taken your first steps toward becoming a world-class RL empiricist! Let's put some of our learnings into practice. In this section we attempt to reproduce a well-known result from the literature, along the way evaluating the original design choices and suggesting alternatives. In the end, we end up with very different results!

This case study documents our attempt to recreate the experiments of the Soft Actor-Critic (SAC) paper (Haarnoja et al., 2018). SAC is an actor-critic algorithm that works in the maximum-entropy RL framework and has been shown to perform well on continuous control tasks and even robotic control (Haarnoja et al., 2018, 2019). We have chosen to outline how improper empirical practices misled the conclusions found by Haarnoja et al. (2018) because SAC is both popular and widely used.

We attempted to reproduce the comparison of SAC and Deep Deterministic Policy Gradient (DDPG) on the Half-Cheetah environment from AIGym. For the SAC implementation,

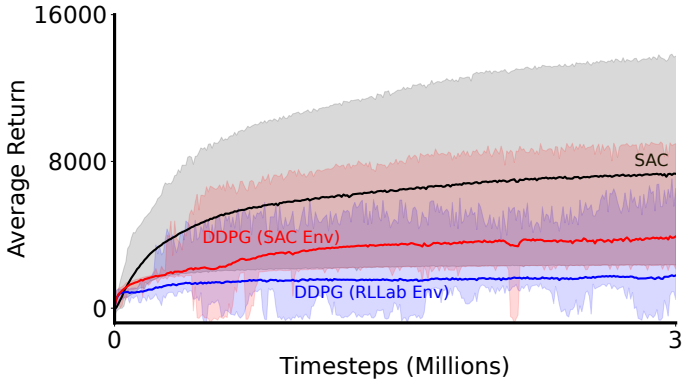


Figure 13: Our first attempt to recreate the experiments in the original SAC paper. In contrast to the original work which used only 5 runs, we use 30 runs. The variance in performance is higher and the average performance lower than reported in the original work. Solid lines denote average performance, and shaded regions denote minimum and maximum performance.

we used the original SAC codebase[18] (SAC-CB) with the tuned hyperparameters outlined by Haarnoja et al. (2018). For the DDPG baseline, we used the RLLab codebase[19] (RLLab-CB) with the hyperparameters set to the defaults in the RLLab codebase, except that the batch size, replay buffer capacity, and network architectures were adjusted to match those used by SAC (Duan et al., 2016). The SAC paper did not outline the exact hyperparameters used for DDPG, but their github repository left some hints on the configuration of the DDPG baseline as well as the implementation used[20]. Two different environment wrappers for OpenAI Gym exist in these codebases, one in RLLab-CB and one in SAC-CB. Which wrapper Haarnoja et al. (2018) used for their experiments with DDPG is unclear, so we use both for DDPG. For SAC, we assume the wrapper in SAC-CB was used.

Figure 13 shows the mean learning curves with shaded regions as minimum and maximum performance. We would like to highlight the large difference between the two DDPG results, only different due to using a different environment wrapper. We identified a possible bug in the environment wrapper for RLLab-CB, causing this wrapper to handle episode cutoffs improperly. Some algorithms in RLLab-CB can bootstrap on episode cutoffs. Such a practice considers the final state due to the episode cutoff as a terminal state due to reaching a goal —considering the cutoff state to have a value of 0 in the TD-error. Even if this functionality is explicitly turned off, this incorrect bootstrapping can still occur[21]. We cannot know for sure, but DDPG may have had a significant disadvantage in the original experiments. In Figure 13, the line labelled *DDPG (RLLab Env)* does suffer from this innocuous bug while the line labelled *DDPG (SAC Env)* does not.

18. The original SAC codebase can be found at `https://github.com/haarnoja/sac`
19. The RLLab implementation can be found at `https://github.com/rll/RLLab`
20. See `https://github.com/rail-berkeley/softlearning/issues/27`. It is mentioned that the original paper used the RLLab implementation of DDPG with similar hyperparameter settings as SAC where applicable. The RLLab implementation uses OU noise for DDPG.
21. This happens if the wrapped AIGym environment uses a number of steps per episode less than or equal to the number of steps per episode in the RLLab-CB environment wrapper.

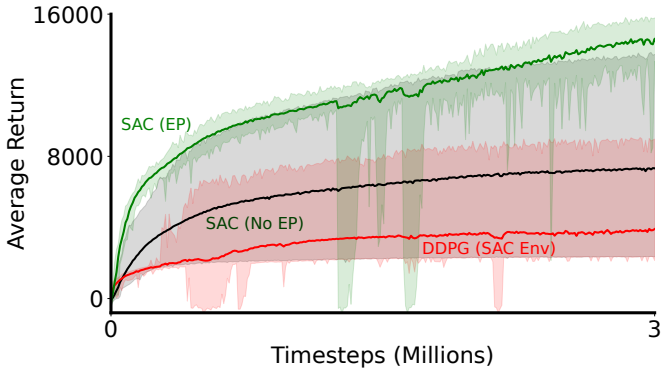


Figure 14: Our second attempt to recreate the experiments in the original SAC paper. We again use 30 runs. The mean performance attained by SAC (EP) closely matches that reported in the original work, but the variance in performance is again higher. This is likely due to the fact that we used more seeds than the original work to evaluate the performance of SAC.

In Figure 13, the mean performance of SAC over 30 runs is lower than that reported in the original paper, and the error bars here are larger than those reported in the original paper. Next, we examined the code-base more carefully, to understand this discrepancy.

In our previously described experiment, we attempted to reproduce the results of Haarnoja et al. (2018) using the experimental procedures described in the paper alone. Yet, several implementation choices in the code-base were not reported in the paper. First, the default implementation of some policies in SAC-CB use regularization (e.g, Gaussian policies). Second, many code examples in SAC-CB normalize actions to stay within the environmental action bounds. Finally, several code examples in SAC-CB use an initial random exploration phase — actions are sampled from a uniform distribution over actions for the first 10,000 steps. Policy regularization, action normalization, and random initial exploration are not reported in the paper. Regularization and action normalization do not affect the squashed Gaussian policy implementation in SAC-CB, so we expected the most likely culprit for the disparity in performance reported by Haarnoja et al. (2018)—and that which we could reproduce—was the initial random exploration phase.

We re-ran the previous experiment using an initial exploration phase of 10,000 steps. After this initial phase, action selection once again became on-policy, with actions selected according to the policy learned by SAC. From here on, we refer to an algorithm $\mathcal{A}$ with this initial exploration phase as $\mathcal{A}$ (*EP*) and without this initial exploration phase as $\mathcal{A}$ (*No EP*). Figure 14 shows the learning curves over 30 runs for this additional variant of the SAC algorithm, SAC (EP). The mean performance of this variant closely matches that reported by Haarnoja et al. (2018), although the variability in performance is noticeably higher. This is likely due to the fact that we used 30 seeds while the original work used only 5.

Finally, previous work found seed optimization in the RLLab code-base, meaning that results are reported by sweeping over seeds and reporting performance only for the best seeds (Islam et al., 2017). This seed optimization code is compatible with SAC-CB as well. As a final attempt to reproduce the results of Haarnoja et al. (2018), we used seed optimization in

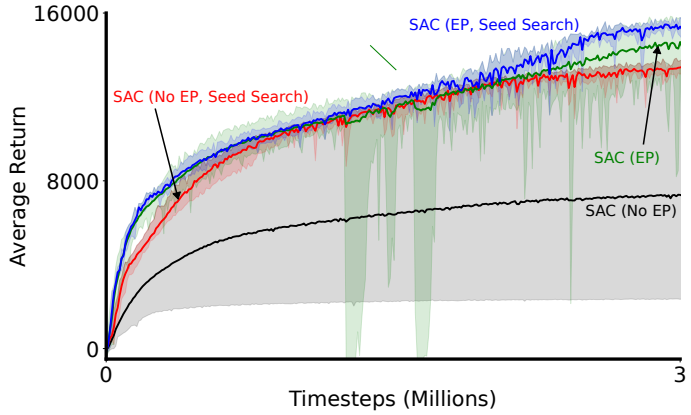


Figure 15: When searching seeds, we were able to achieve average performance and shaded regions much closer to that reported in the original SAC work. Solid lines denote average performance, and shaded regions denote minimum and maximum performance.

the hyperparameter tuning process. **As a note, this is bad practice; we only conduct seed search in the name of reproduction.** In Figure 15, where we chose the best 5 seeds of 30 for each algorithm, the results more closely match those from the original paper. In particular, SAC (EP) with seed optimization most closely matches the results reported by Haarnoja et al. (2018).

We now turn to running an experiment that more closely resembles the principles laid out in this document, especially with respect to reporting performance of *tuned baselines*. Although we do not know how Haarnoja et al. (2018) tuned the baseline algorithms in their experiments, we found that the performance of DDPG on Half Cheetah was under-reported in this work. In the experiments here, we use the tuned hyperparameters for DDPG as reported by SpinningUp baselines[22]. Since Gaussian noise is known to outperform OU noise in some cases (Fujimoto et al., 2018)[23], we use uncorrelated, unbounded Gaussian noise for action exploration in DDPG instead of OU noise. Furthermore, we try both SAC and DDPG with an exploration phase at the beginning of the experiment, where an action is drawn uniformly randomly for the first 10,000 steps. Similarly to previous experiments, we use the tuned hyperparameters reported by Haarnoja et al. (2018) for SAC.

Figure 16 shows the mean learning curves with 95% bootstrap confidence intervals for this tuned version of DDPG and SAC over 30 runs. By tuning the DDPG baseline, we achieved significant improvements in performance over what was reported in the original SAC work. It seems DDPG (No EP) is competitive with and likely better than SAC (No EP) on Half Cheetah. Furthermore, we see that simply using an initial exploration phase can significantly improve the performance of both algorithms on Half Cheetah.

Figure 16 provides one final lesson – perhaps the most important. Unintentionally letting bias creep into experiments is easy. The experimenter must work hard to treat baselines fairly in order to avoid creating misleading results.

22. See https://spinningup.openai.com/en/latest/spinningup/bench.html
23. The benefits of Gaussian noise were published in parallel with the SAC paper in 2018. It is therefore reasonable to assume that the authors of SAC did not know the advantages of Gaussian noise.

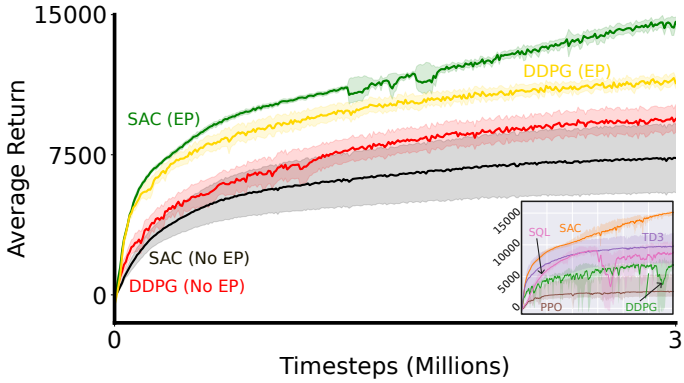


Figure 16: Our attempt to run an experiment that closely resembles the principles laid out in this document, particularly with respect to tuning baselines. For reference, the original experiments Haarnoja et al. (2018) conducted with SAC on Half Cheetah have been inset. DDPG has been tuned here for Half Cheetah. See text for details.

8 Conclusion

The goal of this document was to provide a comprehensive overview of important empirical design decisions in reinforcement learning. The style is educational, with a focus on clear examples and conceptual reasoning for making sound decisions. In the first few sections we focused on outlining best practices for evaluating reinforcement learning algorithms. We then revisited an existing result, as a case study, to highlight different conclusions in light of these best practices. Much of the work focused on what to do, with some comments about what to avoid. For clarity, therefore, we concluded the work with a more explicit list of common errors to avoid. We hope for this document to particularly help newcomers to reinforcement learning, but also to provide novel perspectives for any reinforcement learning empiricists.

References

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C. Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. In *International Conference on Machine Learning*. PMLR, 2018.

Leemon Baird. Residual Algorithms: Reinforcement Learning with Function Approximation. *Machine Learning Proceedings*, 1995.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 2013.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Maxime Chevalier-Boisvert, Lucas Willems, and Pal Suman. Minimalistic Gridworld Environment (MiniGrid). Farama Foundation, 2018.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments. *arXiv:1806.08295*, 2018.

David Colquhoun. The reproducibility of research and the misinterpretation of p-values. *Royal society open science*, 2017.

William C Dabney. *Adaptive Step-Sizes for Reinforcement Learning*. PhD thesis, 2014.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*. PMLR, 2016.

Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 1986.

Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *International Conference on Machine Learning*, 2018.

Sina Ghiassian, Andrew Patterson, Shivam Garg, Dhawal Gupta, Adam White, and Martha White. Gradient Temporal-Difference Learning with Regularized Corrections. *International Conference on Machine Learning*, 2020.

Dibya Ghosh and Marc G. Bellemare. Representations for Stable Off-Policy Reinforcement Learning. *International Conference on Machine Learning*, 2020.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *International conference on machine learning*, 2018.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications, 2019.

Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 2021.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. *AAAI*, 2018.

J. Fernando Hernandez-Garcia and Richard S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the DQN target. *arXiv preprint arXiv:1901.07510*, 2019.

John N. Hooker. Testing heuristics: We have it all wrong. *Journal of heuristics*, 1995.

Ehsan Imani, Eric Graves, and Martha White. An off-policy policy gradient theorem using emphatic weightings. *Advances in Neural Information Processing Systems*, 2018.

Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. *arXiv:1708.04133 [cs]*, 2017.

Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, 2011.

Scott M. Jordan, Yash Chandak, Daniel Cohen, Mengxue Zhang, and Philip S. Thomas. Evaluating the Performance of Reinforcement Learning Algorithms. *International Conference on Machine Learning*, 2020.

Seungchan Kim, Kavosh Asadi, Michael Littman, and George Konidaris. DeepMellow: Removing the Need for a Target Network in Deep Q-Learning. *International Joint Conference on Artificial Intelligence*, 2019.

J Z Kolter. The Fixed Points of Off-Policy TD. *Advances in Neural Information Processing Systems*, 2011.

Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study. *Advances in neural information processing systems*, 2018.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 2018.

Ashique Mahmood. Incremental off-policy reinforcement learning algorithms. 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. 2013.

Andrew William Moore. *Efficient Memory-Based Learning for Robot Control*. PhD thesis, University of Cambridge, 1990.

Johan S. Obando-Ceron and Pablo Samuel Castro. Revisiting Rainbow: Promoting more insightful and inclusive deep reinforcement learning research. *International Conference on Machine Learning*, 2021.

Andrew Patterson, Victor Liao, and Martha White. Robust losses for learning value functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022a.

Andrew Patterson, Adam White, and Martha White. A Generalized Projected Bellman Error for Off-policy Value Estimation in Reinforcement Learning. *Journal of Machine Learning Research*, 2022b.

Thomas Sellke, M. J. Bayarri, and James O. Berger. Calibration of ρ values for testing precise null hypotheses. *The American Statistician*, 2001.

Alexander L. Strehl and Michael L. Littman. An analysis of model-based interval estimation for Markov decision processes. *Journal of Computer and System Sciences*, 2008.

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 1996.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

Richard S. Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. *International Conference on Machine Learning*, 2009.

Richard S Sutton, A Rupam Mahmood, and Martha White. An Emphatic Approach to the Problem of Off-policy Temporal-Difference Learning. *Journal of Machine Learning Research*, 2016.

István Szita and András Lorincz. The many faces of optimism: A unifying approach. In *Proceedings of the 25th International Conference on Machine Learning*, 2008.

Adrien Ali Taïga, William Fedus, Marlos C. Machado, Aaron Courville, and Marc G. Bellemare. Benchmarking Bonus-Based Exploration Methods on the Arcade Learning Environment. *Exploration in Reinforcement Learning Workshop, ICML*, 2019.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 1995.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*. IEEE, 2012.

J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 1997.

Hado van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep Reinforcement Learning and the Deadly Triad. *arXiv:1812.02648*, 2018.

Charles V. Vorhees and Michael T. Williams. Morris water maze: Procedures for assessing spatial and related forms of learning and memory. *Nature protocols*, 2006.

Han Wang, Erfan Miahi, Martha White, Marlos C. Machado, Zaheer Abbas, Raksha Kumaraswamy, Vincent Liu, and Adam White. Investigating the Properties of Neural Network Representations in Reinforcement Learning, 2022.

Ronald L. Wasserstein and Nicole A. Lazar. The ASA Statement on p-Values: Context, Process, and Purpose. *The American Statistician*, 2016.

Adam White and Martha White. Investigating practical linear temporal difference learning. *International Conference on Autonomous Agents and Multi-Agent Systems*, 2016.

Shimon Whiteson, Brian Tanner, Matthew E. Taylor, and Peter Stone. Generalized domains for empirical evaluations in reinforcement learning. *Workshop on Evaluation Methods for Machine Learning at ICML*, 2009.

Shimon Whiteson, Brian Tanner, Matthew E. Taylor, and Peter Stone. Protecting against evaluation overfitting in empirical reinforcement learning. *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, 2011.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*. PMLR, 2020.

Appendix A. Summary of Contributions

The primary contribution is to systematically catalog folk wisdom in RL experiments, and reasons for choices that might not be written elsewhere, as well as to rethink some of our empirical practices by systematically writing them down. We realized for ourselves that some of our empirical practices are insufficiently justified, and that we should consider alternatives. As such, this document is written educationally, documenting this reflection. But, because we had to answer questions along this journey, it does also contain a variety of new results, beyond just conceptually reasoning about our choices. It can be hard to find those new results, nestled amongst this conceptual reasoning. In this section, we summarize a list of novel proposals and empirical findings in this work.

The novel methodological proposals include the following.

1. (Section 2.4) Tolerance intervals with median performance can better highlight the instability of an agent within a run, and variability across runs, than our standard approach of using means and confidence intervals (see Figure 5c versus Figure 6).

2. (Section 3.2) An efficient algorithm for estimating idealized performance (in Algorithm F.3), fixing issues with maximization bias and the typical two-stage approach uses for hyperparameters.

3. (Section 4.4) Controlling the source of randomness to reduce variance, without introducing bias. This included (a) looking at differences in algorithm performance, to enable paired testing for algorithms and (b) separating the random seed for the agent and environment to better control sources of variability (see Example 4.4).

The novel empirical findings include the following.

1. (Section 2) The performance distributions for our agents can be skewed and bimodal (DQN on Mountain Car in Figure 4, DQN on PuddleWorld in Section 2.6).

2. (Section 2.7) Episode cutoffs can have a large impact on algorithm performance, as demonstrated in Figure 8a. This result highlights that the short cutoffs used are making environments easier for our agents.

3. (Section 2.6) A demonstration that common confidence interval approaches can fail to capture the mean, for the long-tailed performance distribution of DQN on PuddleWorld (Figure 7).

4. (Section 3.1) Momentum likely does not resolve issues with divergence in TD, as a TD with momentum still diverges on Baird's counterexample (see Example 3.1).

5. (Section 3.2) If we run 1000 experiments of DQN on the Mountain Car domain and sweep stepsizes, target network refresh rates, and replay buffer sizes for every experiment, then approximately 96% of these experiments will over report the average performance for the best hyperparameter configuration among those tested.

6. (Section 4.1) DQN with default hyperparameters can fail on even simple environments; with a small amount of tuning, it goes from failure to very good performance on Lunar

Lander (see Example 4.1). This demonstrated the issue with using untuned baselines and making conclusions about the utility of a new approach.

7. (Section 7) A thorough case study applying the methodologies laid out in this work. This case study highlighted issues with reproducing a previous result, issues in that previous experimental design that lead to misleading conclusions and a final result that provided a more clear empirical picture of those algorithms (Soft Actor-Critic and DDPG).

Appendix B. Further experimental details

B.1 ESARSA and the Maze Gridworld

For this experiment, we combine the Expected SARSA algorithm (ESARSA) using an ϵ-greedy policy both as the bootstrapping target and as the behavior policy. The agent uses tile-coded features mapping the (x, y)-coordinates within the gridworld to a large binary feature vector. The state, action value function estimate is a linear function of the tile-coded features.

Hyperparameter	Value
Tiles	4
Tilings	8
Stepsize	0.1
ϵ	0.2
Experiment length	30k steps
γ	0.99

Appendix C. Computing tolerance intervals

Computing a tolerance interval is simple and is independent of the underlying distribution of the data. We define an (α, β)-tolerance interval as an interval that captures β proportion of future samples (e.g. $\beta = 0.9$) with a nominal error rate of α (e.g. $1-\alpha = 0.95$). Intuitively, to capture β proportion of future samples, we might think to report upper and lower percentiles of the sample data, such that the percentiles symmetrically capture a β proportion of samples. In the case that $\beta = 0.9$, this would correspond to an upper percentile $u = 0.95$ and a lower percentile $l = 0.05$. A tolerance interval takes this a step further by including an uncertainty correction—when we have received a small number of samples, how do we know the 95th percentile of the samples corresponds to the 95th percentile of the true distribution? Tolerance intervals, then, add a slight pessimism by widening the interval based on the number of observed samples. As we observe more samples, the uncertainty decreases and the interval approaches the naive percentile-based approach.

To compute the uncertainty corrected percentiles, we make use of the inverse CDF of the binomial distribution. We want to ask the question: *for each sample in our dataset, does this sample lie within the middle β proportion of the distribution?* Our success rate for the binomial distribution, then, is β and our accepted error rate is α. The inverse CDF provides the number of samples ν that do not belong to the middle β proportion of the distribution. We distribute ν evenly across the top and bottom of the distribution, receiving indices of

the sorted data $l = \frac{\nu}{2}$ and $u = n - \frac{\nu}{2}$. Note that when ν is odd, these indices will no longer be integers. A common practice is to interpolate evenly between the adjacent indices, or to alternatively take the floor of the lower index l and the ceiling of the upper index u. The interpolation approach generally provides more accurate bounds for smaller sample sizes.

Appendix D. More about hyperparameter selection

In this section, we provide a more in-depth discussion on hyperparameter selection. It is a topic that could fill an entire textbook, and so we opted to keep only the most basic information in the main body of this document. Here, we dive a bit deeper, to highlight a few other more advanced points.

D.1 Understanding hyperparameter sensitivity for multiple hyperparameters

Characterizing hyperparameter sensitivity with only one hyperparameter is relatively straightforward; it becomes more complex with multiple hyperparameters. The issues are that (1) there is a potentially combinatorial explosion, (2) there are likely interactions between hyperparameters and (3) visualization becomes more difficult. As yet, there is no consensus strategy for understanding the performance of a partially-specified algorithm with multiple unknown hyperparameters, but we discuss a few here.

A basic strategy—that only shows variability across hyperparameters rather than interactions between them—is to use violin plots. We visualize this in Figure 17, for multiple algorithms. The idea is to select a set of hyperparameter settings, compute performance for each setting, and report the distribution of performance over all settings. This allows us to compare algorithms with different hyperparameters, and still ensure they get the same number of hyperparameter settings.

The one important nuance here is that, unlike sensitivity plots, selecting a wider range of hyperparameters can be misleading. Visually, violin plots encourage us to asses variability across hyperparameters. If we set the range to be too wide for one algorithm (say our competitor), then it may look sensitive simply because I selected an unreasonable range. Fairly selecting ranges for the hyperparameters should be easier than selecting the hyperparameters themselves, though, and violin plots are a useful tool when assessing performance with multiple unknown hyperparameters.

To better understand relationships between hyperparameters, one plausible strategy is to collect a data set where hyperparameters are the independent variables and performance values are the dependent variables. We can then fit a model to this dataset to describe relationships between hyperparameters as well as their relationship to performance. Using such a strategy and a linear regression model, we could find—for instance—linear correlations between multiple hyperparameters. This strategy is related to the AutoML and Bayesian optimization communities, though with the caveat that our interest is in the model itself while these communities generally use the model as a means to perform optimization (He et al., 2021).

We could additionally adopt a classic randomized-control trial approach to understanding the impact of one hyperparameter while others are left as unknowns. With this strategy, we would test the hyperparameter of interest at multiple predefined levels (a hyperparameter sweep) while all other hyperparameters are treated as nuisance variables and, in an RCT

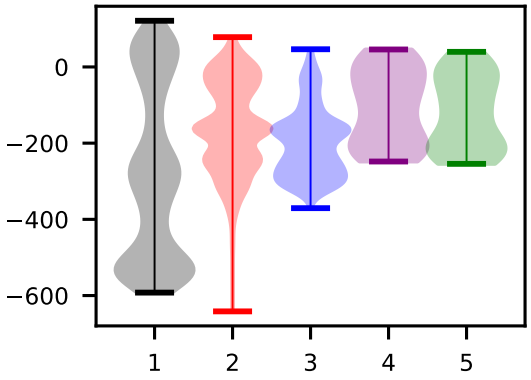


Figure 17: Visualizing the hyperparameter sensitivity of several algorithms on Lunar Landar.

experiment, nuisance variables are randomized over. This strategy is highly related to a recently proposed approach (Jordan et al., 2020), with the slight difference that Jordan et al. (2020) treats *all* hyperparameters as nuisance variables while we allow one to be controlled.[24]

The design methodologies discussed in this section are constantly changing as the field matures. Currently, we discuss treating hyperparameters as unknowns while performing scientific analysis and using statistical reasoning to understand their impact. However, as better hyperparameter optimization methodologies are developed and widely adopted, it will no longer be true that hyperparameters are strictly unknowns. Consider the case of supervised learning. Hyperparameters may be initially unknown but a procedure for selecting them *is* known: cross-validation. As such, it is generally far more accurate and less error-prone to use cross-validation to set hyperparameter values as part of the scientific analysis instead of treating them as nuisance. Lacking such a well-adopted strategy in reinforcement learning, in the most general case we should continue to consider hyperparameters as unknowns.

It is important also to note that tuning over hyperparameters is a luxury we have in our experiments, rather than a general-purpose algorithm to set hyperparameters. We are not advocating here for the use of hyperparameter sweeps to select hyperparameters, as this is rarely a viable option for real-world problems, unless the problem is a simulator.[25] Instead, the goal of hyperparameter sweeps in our experiment is to get insights into hyperparameter sensitivity, and to identify algorithms that are generally performant and relatively insensitive to their hyperparameters and so more suitable for deployment.

D.2 Developing algorithms and avoiding misleading yourself

Recall we had two possible goals in conducting an experiment: (1) understanding performance with respect to an algorithms hyperparameters, and (2) optimizing the hyperparameters for

24. In fact, RCT experiments can be easily extended to allow multiple hyperparameters to be controlled. Such an experiment design is often called *response surface methodology* or *factorial design*.

25. There are many groups pursuing algorithm development in reinforcement learning to solve simulated problems, like games or hard search problems. Our goal is to help empirical design more generally in reinforcement learning, not just for this more restricted problem setting, and so we do not consider specific approaches that can exploit simulators.

a specific problem setting. In this section we discuss something sort-of in between: selecting hyperparameters when you are developing a new algorithm. Similar to the last section, your goal here will be to understand the impact of your new hyperparameters. The primary difference, however, is that you will likely want to iterate to improve your algorithm. Further, you might be building on other algorithms that already have hyperparameters, and your modification to the algorithm might interact with those existing hyperparameters.

> **Example: Your new algorithm**
> In some preliminary experiments, you realize that DQN is quite sensitive to its target network update frequency. These exploratory observations inspire you come up with a new algorithm to adapt the update frequency during learning. Your algorithm, however, has a new hyperparameter. Naturally, you want to see the behavior of your algorithm under ideal conditions, where you look at performance for a nearly optimal hyperparameter setting, found either use sweeps or other (smarter) hyperparameter optimization approaches.

This is a reasonable first step—while developing an algorithm we often want to know "does this work *at all*?"—however we cannot stop our investigation here. First, it is likely you can improve on DQN by using its same hyperparameter settings for the environment, and tuning over this additional hyperparameter. You may falsely conclude that your algorithm provides benefits, when in actuality the main affect was the ability to optimize over an additional scalar. Second, this modification to DQN, and your new hyperparameter, might interact with existing hyperparameter choices in an unexpected ways. In fact, such a result has recently been shown for GANs; many modern GAN architectures provide little or no improvement over a sensibly tuned baseline (Lucic et al., 2018).

Instead, we should immediately perform the follow-up step: examine the performance of your algorithm for multiple values for the new hyperparameter. This is a tricky and nuanced procedure, we cover many of the common cases above in Section 3.1. During the algorithm development cycle, it can be expensive to perform repeated sensitivity studies. We strongly recommend that this is done on smaller, meaningful problem settings where simulation is cheap. Such studies are often called *pilot studies* and are generally used to inform the design of more complete studies later in the process.

D.3 Why can't we just use cross-validation?

A natural question is why we cannot handle hyperparameters in the same way they are handled in supervised learning: internal cross-validation. In this section, we explain why internal cross-validation does not directly apply to reinforcement learning.

Let us start with a brief refresher of cross-validation. Imagine you have a training set and learn a function f with any regression approach. You want an estimate of its generalization error, but do not want to split up your already small training set into a training and testing set. Instead, you can use cross-validation. The idea is to partition the dataset into k folds, and train on each subset of $k-1$ folds and test on the remaining fold. This procedure generates k functions $f_1, \ldots, f_k$ with corresponding estimates of error $e_1, \ldots, e_k$. The average of these k errors provides a reasonable estimate of the generalization error of f, even though they are estimates of error for different functions. *The primary role of cross-validation is to estimate the generalization error of a function*, without needing a hold-out set.

The same idea can be used to select hyperparameters, in a procedure called internal cross-validation. The reason cross-validation can be used for this purpose is that it allows us to estimate the generalization error of each function learned with different hyperparameters. The goal is to pick the function with the best generalization error. The algorithm needs to both (a) specify its own hyperparameters and (b) learn its weights with regression. We can see this as an expanded learning problem, with cross-validation used as the algorithm to identify the hyperparameters. We can use cross-validation on the given training set to evaluate each hyperparameter setting and pick the one with lowest error. This complete algorithm A inputs a training set and outputs a function f. We might use external cross-validation to evaluate A, which itself uses internal cross-validation. The external cross-validation gives an estimate of performance, before deploying the model f learned by A on the entire training set.

More simply all of the above could have been described by assuming we split our dataset into training and validation. The validation set allows us to evaluate each hyperparameter, because it allows us to evaluate the generalization error of the function learned with that hyperparameter. After identifying a good hyperparameter using this approach, one would then retrain the function on all the data before deploying. The training-validation split approach is simpler, but does not provide as good an estimate of the generalization error of the function trained on all the data, especially if we have a smaller amount of data.

This idea does not directly extend to hyperparameter selection for online reinforcement learning, because we evaluate a learning algorithm rather than a learned model. We can only see performance of the learning algorithm once it is in deployment (testing). There is no separate training phase, nor training data, for the online reinforcement learning setting. Overall, there is no obvious, out-of-the-box way to directly use cross-validation for hyperparameter selection.

D.4 Cross-validation-like procedures for the hyperparameter optimizer

We can, however, consider strategies inspired by these ideas from cross-validation. We propose one such view below. Imagine we care about how our agent performs on a class of environments $\mathcal{E}$—none of the below will make any sense if we only care about performance in one environment. We only receive a subset of these environments, $\mathcal{E}_{\text{given}}$; let's say we have n such (training) environments. The goal is to answer: how will my (fully-specified) algorithm perform on $\mathcal{E}$, given only access to $\mathcal{E}_{\text{given}}$?

If our algorithm has no hyperparameters, then we can run this fully-specified algorithm in our training environments, to get samples of performance $p_1, \ldots, p_n$, and take the average. If $\mathcal{E}_{\text{given}}$ is a random subsample, then this sample average estimate is an unbiased estimate, and with big enough n a relatively good reflection of performance in expectation across $\mathcal{E}$. In fact, it is more accurate to think of $\mathcal{E}_{\text{given}}$ as a testing set, because we are evaluating our algorithm using this set rather than using it for training. No training occurs here, because our algorithm does not transfer any agents from these environments. It learns from scratch on any new environment in $\mathcal{E}$.

If we have hyperparameters to specify in our algorithm, then we can consider how to use $\mathcal{E}_{\text{given}}$ to specify these hyperparameters. Now there is a notion of using $\mathcal{E}_{\text{given}}$ like a training set, but not for our algorithm, but rather the hyperparameter optimizer H. Each environment is like a training point. The hyperparameter optimizer H can train on these

environment training points, to output a proposed set of hyperparameters h: $H(\mathcal{E}_{\text{training}}) = h$. These hyperparameters are then deployed, and we hope that they generalize to the larger class of environments.

The ability to generalize well depends on the quality of the set $\mathcal{E}_{\text{given}}$—just like in supervised learning—and also on the hyperparameter optimization algorithm H. For example, H could be a simple grid search or H could be a Bayesian optimization algorithm. Likely these two algorithms will identify different hyperparameters using the set $\mathcal{E}_{\text{given}}$, and so will result in different generalization performance. A grid search algorithm estimates the performance of each hyperparameter choice by computing per-environment performance over multiple runs in each $\mathcal{E}_{\text{given}}$, and then aggregating that performance into one number. It deploys the hyperparameter λ_{all} with the best cross-environment performance number. If we had to map to the supervised learning setting, the λ_{all} is like the deployed function f above.

We can use a similar idea to (external) cross-validation to estimate how well these hyperparameters λ_{all} might generalize. How do we know if λ_{all} generalizes well to other environments? We can keep a hold-out set of environments to test generalization performance. We split $\mathcal{E}_{\text{given}}$ into training and testing environments, and only give the training environments to H to produce h. Then we can test the fully-specified algorithm, with hyperparameters h, in the test environments to get a sense of performance.

This is where the ideas behind cross-validation help. Just using training-testing splits is "data" inefficient: we "learn" λ_{all} on an even smaller training set, potentially resulting in a λ_{all} that has worse generalization performance. Instead, we can estimate how well λ_{all} might do, by looking at generalization performance of all the hyperparameters trained on subsets of the environments.

More specifically, imagine we have n environments. We run H on all n to get λ_{all}. Now we use cross-validation to evaluate λ_{all}. We find the λ_1 that is the best hyperparameter across all the environments except environment 1 in $\mathcal{E}_{\text{given}}$, and so on. The generalization performance of λ_1 is estimated using performance on environment 1, to get p_1. Then an estimate of generalization performance for λ_{all} is $\frac{1}{k}\sum_{i=1}^{k} p_i$. This does not actually change what hyperparameters are deployed: we are still deploying λ_{all}. It just lets us get an estimate of the quality of these before deployment.

Note that this procedure has never been used, to the best of our knowledge, and so it is not clear that it enjoys the same nice properties as cross-validation. It might have unexpected sources of bias. We are not advocating that we use the above procedure, but rather trying to bring clarity by showing potential connections to standard algorithms in supervised learning.

D.5 Even more about treating the hyperparameter optimizer as the algorithm

The above was only using external cross-validation to estimate generalization performance. But if the hyperparameter optimizer itself has hyperparameters—let's call them hyper-hyperparameters for lack of a better term—then we can exploit the fact that the cross-validation-like approach above estimates generalization error. We want to select the hyper-hyperparameters such that they produce hyperparameters λ_{all} that generalize best. For example, grid search might have a hyperparameter that is the number of runs or number of steps of interaction in the environment. These could be chosen using an internal cross-validation approach.

In summary, the typical internal cross-validation approach from supervised learning could be used in reinforcement learning to set the hyper-hyperparameters of our hyperparameter optimizer. There is no sensible mapping where internal cross-validation itself is used to select the hyperparameters of the reinforcement learning algorithm. If our hyperparameter optimizer does not have any hyper-hyperprameters, then we simply run the hyperparameter optimizer on the training set and deploy the hyperparameters that are found. We can use external cross-validation to estimate the quality of these hyperparameters. Otherwise, if the hyperparameter optimizer has 5 different hyper-hyperparameter settings that could be used, then we use internal cross-validation to pick amongst the hyper-hyperparameters.

It is important to point out that we do not truly have training sets (of environments). The standard assumption in supervised learning is that the training set is representative of the testing set.[26] In reinforcement learning, if we are deploying an algorithm to control a physical system, we do not have a set of other physical systems where we can evaluate the agent first. A practitioner might actually try to craft this set themselves. Maybe they scour the literature for environments that resemble their environment, such as environments in Mujoco or simulators for other real-world problems. They might then evaluate their algorithm—and tune the hyperparameters—in those environments. However, such a procedure has not been explicitly proposed in the literature, nor is it standard practice. It would be interesting to explore such ideas, and other standard hyperparameter selection approaches, to get to a similar place that supervised learning is in with cross-validation.

Appendix E. Aggregating environments

Individual environments can also be grouped, to provide a *macro-environment* for which we examine performance of an algorithm. We run the algorithm on each environment in the group, but consider the aggregate performance across environments. By thinking of this environment grouping as a macro-environment, it encourages curation of reasonable groupings to test different properties of the algorithms. If we group environments with low-dimensional inputs and have a separate group for image-based inputs, then we can test the algorithms separately on these two macro-environments to understand behavior on MDPs with these two different properties. Grouping environments, therefore, can improve on benchmarks by making experiments more issue-oriented.

It also has the additional benefit that we can test the algorithm in a larger set of environments, without having to do as many runs for each environment. If there are 5 environments in the macro-environment, it may be sufficient to use 10 runs for each, giving a total of 50 runs for aggregate performance in the macro-environment. Because we are not making claims about performance within each environment, fewer runs per environment are acceptable. It can still be reasonable to visualize individual runs in the environments, for qualitative insights such as those related to instability within runs. Performance plots, however, should only be reported for the macro-environment. We visualize what performance could look like, in Figure 18 for each environment when there are too few runs, resulting in

26. Of course, there are settings with distribution shifts, between training and test. But, there is still an assumption that a training set exists and has some useful connections to the deployment (test) scenario. We have no clear mechanism to obtain training sets yet, in reinforcement learning, nor is it even clear that we should.

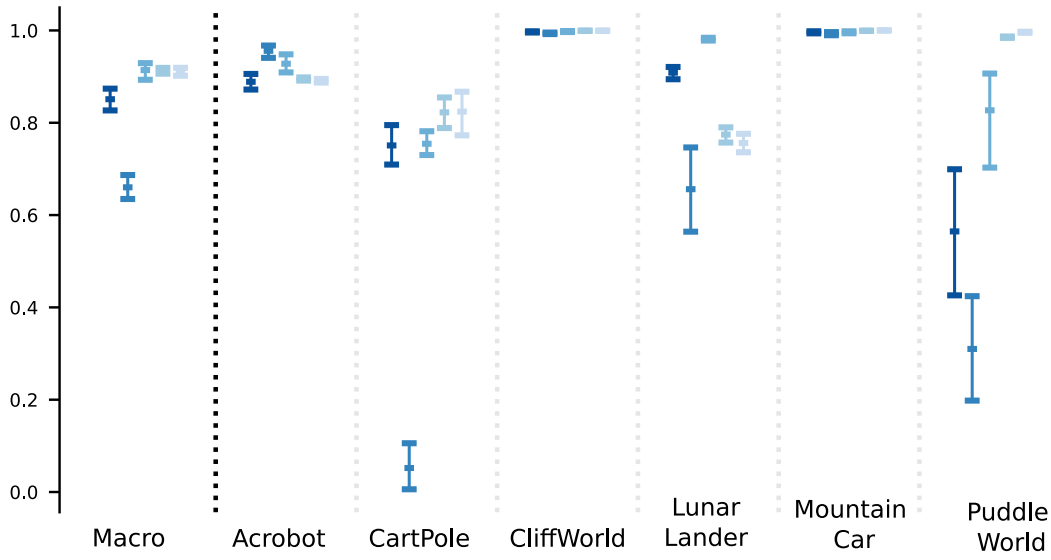


Figure 18: Bootstrap confidence intervals around average performance across tasks (Macro environment) versus for each independent task. It is difficult to assess an ordering of algorithms when investigating environments individually, however when measuring performance across environments jointly a more clear ordering emerges. Also by combining data from all environments, the confidence intervals for the Macro environment are much more narrow.

highly-overlapping errors bars, whereas aggregate performance across runs provides clear differences between the two algorithms.

Reporting such aggregate performance, however, raises the issue of how to ensure performance is normalized across environments. Such normalization could be hand-designed by the experimenter, to ensure each macro-environment has a sensible distribution. For example, the experimenter may know the optimal expected return G^* and worst-case return G^- in each environment. The observed returns G_t could be normalized between 0 and 1 using

$$\frac{G_t - G^-}{G^* - G^-}.$$

If this number is 1, the return is optimal; if it is zero, the return was the worst-case. Special care should be taken when using ratio-based scaling methods, as these can be sensitive to relative size of the scale (Fleming and Wallace, 1986; Jordan et al., 2020).

In addition to creating such macro-environments, there are already several environment suites that have been designed to have related but different environments. Examples include Atari (Mnih et al., 2013), Metaworld (Yu et al., 2020) and MiniGrid (Chevalier-Boisvert et al., 2018). Earlier work also considered minor variations of the same problem, such as Mountain Car with different perturbations to transitions, observation transformations and start-state conditions (Whiteson et al., 2011).

Appendix F. Understanding our agents with multiple evaluation metrics

For most of this work we have been relatively agnostic to the choice of evaluation metric. In the initial section we discussed the typical evaluation metric: the return. After seeing how to measure this evaluation metric in one run for one agent, we discussed strategies to assess an algorithm across multiple runs by aggregate this evaluation metric across runs. These aggregation strategies were generic, and apply to other evaluation metrics.

In this section, we discuss alternative evaluation metrics that could help you assess an algorithm. In some cases, the expected return is not a suitable evaluation metric. Instead, for example, you might want to know if your agent maintains a certain level of performance once reaching that level. More generally, using multiple metrics can provide a more complete picture of the properties of your algorithm. Some metrics may be behavioral, rather than performance-based. For example, it may be useful to understand state-visitation for an agent. We distinguish such *behavioral metrics* from *performance metrics*, and discuss options for both in the next two subsections.

F.1 Measuring the performance of an agent

In order to compare, rank, describe, and tune our algorithms thus far, we have described the performance of agent using the total episodic return obtained by that agent over time. In fact, the overwhelming majority of research in reinforcement learning has focused on describing how well an RL agent solves a given problem. However, there are many possible attributes of our agents which we can measure and many different metrics we can consider for each attribute. A fruitful path towards understanding an agent or an algorithm is developing a multidimensional view of that agent's behaviors and internal processes. There have been countless metrics and attributes studied in the machine learning literature—certainly far more than we could sensibly explore here—however we will describe a few important choices which can serve as a starting point.

Perhaps the most simple deviation from measuring the total episodic return of an agent is applying a weighting to each episodic return based on the number of learning steps within that episode. Consider an environment where an agent spends ten thousand steps to complete the first episodic, such as the Mountain Car environment. When the agent starts the second episode, it has already accumulated ten thousand experiences and has likely performed ten thousand updates to its value function. Comparatively, an agent which solves the first episode in only a few hundred steps, it starts its second episode with far less experience and far fewer updates. Comparing these two agents episode-by-episode means comparing agents with wildly differing amounts of experience, which may not be a meaningful comparison! We could, instead, compare each agent at each update timestep, ignoring episode boundaries. Our performance measure, then would need to assign a performance value at every step. A simple proposal: for each timestep of an episode, record the total return observed for that episode. If an agent takes fifty steps to complete an episode and yields a return of -100 at the end of the episode, then the agent would record a performance value of $[-100, -100, \ldots, -100]$ repeated fifty times—once for every step of the episode.

There are many interesting questions we can ask about the agent which are not directly tied to performance, such as: "what representations are learned by our agent?" Some attributes of interest include the capacity of the representation—what functions can learned?—the

efficiency of the representation—are the duplicate features?—or the amount of interference observed by the representation for each new observation (Wang et al., 2022). Although it may not be immediately clear that these additional measurements correlate with how well an agent solves a task, these can provide distinguishing information between individual agents. Over time, and with many such measurements, we might begin to notice patterns such as: agents with high degrees of interference tend to learn quickly, but fail to adapt to small changes in the environment. These correlations and relationships provide novel avenues for algorithm development.

F.2 Summarizing performance over time

The performance of an agent evolves with time, as do other attributes of our agents; we want our evaluation to capture this. On the other hand, it is likewise challenging to rank, compare, or tune algorithms when our performance metric is a stochastic process or a list of numbers. Quite often, we need to provide a scalar summary which describes the measurements that we have taken over time. A common choice—referred to as area under the curve—is to compute the total amount of an attribute scaled by the number of observations, the average over time.

A persistent challenge in providing a single summary scalar, however, is that nuances in the learning curve will be lost. A time average of the learning curve is akin to fitting a horizontal over the curve, which clearly loses the fact that the curve might increase or decrease steadily with time or might have sharp drops in performance on occasion. Time averages also ignore the fact that learning curves can often be broken into two (or more) phases, generally a rapid upward slope during the early learning process, then a progressive plateau towards the end of learning. Depending on the research question at hand, it can often be desireable to reason about only one of these phases, for instance by computing a time average over the last 10% of the learning curve in order to discuss where the algorithm generally plateaus. There has been some work automatically identifying these phases, by fitting piecewise linear functions to the learning curve (Dabney, 2014, Chapter 3). As usual, there is no right answer, and as the experiment designer you need to make an appropriate choice for your setting.

Once you have isolated the evaluation phase, it may be reasonable to use alternatives to the average to summarize this portion of the curve. One option is to evaluate the worst case performance during evaluation; does your agent consistently perform reasonably well or does it exhibit occasional large drops in performance? This evaluation may be more suitable for settings where it is key to maintain reasonable performance for almost all episodes such as in medical applications where the preference is to ensure most patients get a reasonable treatment, rather than obtaining good outcomes on average across patients. The average could be maximized by having very good treatments for some patients and very poor ones for others. As another example, if the agent is learning to land a helicopter, then we expect that it should never crash the helicopter in the evaluation phase. Reporting worst-case performance in the evaluation phase, rather than average, might better reflect the desired performance for this agent.

One challenge in summarizing learning curves is that there are no agreed-upon definitions for *learning speed* and *stability*. We can provide strategies to operationalize these concepts, however without concrete definitions it can be challenging to ensure our proposed summaries

perfectly reflect these properties. For instance, perhaps we define learning speed as how quickly the agent reaches a reasonable policy. To measure this, we can define a threshold of "reasonable" performance—usually coming from domain knowledge—and measure how many learning steps it takes to cross this threshold. Similarly, we might define stability as staying above that threshold of performance once it has been achieved; measured by counting how often the agent dips below this threshold in the evaluation phase. Does the agent stay above this performance level, or does it often regress and require re-learning?

These particular measures present a challenge, however. What if due to stochasticity our agent manages to cross our designated threshold of performance very early in learning, long before it has actually learned a sensible policy? We might spuriously decide this agent is unstable, when in reality if we had selected a slightly later cutoff point for the evaluation phase we would have seen this agent remain stable. To avoid this potential stochasticity, we can alter our measure to require, say, three consecutive steps above the threshold before we say the agent has crossed from the learning phase to the evaluation phase.

The choice of 3 is not optimal, nor theoretically motivated for either stability or learning speed. Instead, it is a choice motivated by the fact that we would like to be robust to stochasticity, but do not want to use too many consecutive steps as then we are measuring "reached reasonable performance and was stable" rather than "reached reasonable performance". These choices have to be made carefully, and potentially revisited. For this reason, it can be useful to provide summary performance metrics, but also provide learning curves—even showing each run—to allow for more detailed viewing for a reader that would like to dig deeper. Our job as authors is to provide insightful summaries without overloading the reader, so such detailed information may be better included in a supplement.

F.3 Offline returns versus online returns

We have so far exclusively considered online returns. It is common, however, to report offline returns. At each measurement step, we take the current policy and do multiple rollouts in the environment, to estimate it's current expected return. This estimate is used as the performance metric at that measurement step, plotted in the learning curve.

Reporting online returns corresponds to the online learning setting, whereas plotting offline returns corresponds to the pure exploration setting. In the online learning setting, the agent is faced with the exploration-exploitation dilemma. It is being evaluated by how much reward it receives while learning, and so it may want to take the action it currently thinks will achieve most reward (act greedily). But, it also needs to spend some time exploring, to ensure it has not settled on a suboptimal policy and so is missing out on more reward. An agent that balances these well will perform well according to online returns.

For offline returns, the agents behavior during learning is not evaluated. In that sense, it does not face the exploration-exploitation dilemma. Instead, the behavior is faced with the pure exploration question: what actions should it take to learn the target policy (near-optimal policy) on that generated data? The learning curve reflects: if the agent was able to have a pure learning phase for t steps, and then deployed its fixed policy, here is how that fixed policy would perform.

Without looking carefully at a plot and its description, learning curves for these two settings can look similar. But these two settings are fundamentally different. It is important to motivate why you chose online or offline returns in evaluation.

Key insights: understanding agents with multiple evaluation metrics

1. Multi-dimensional analysis is key for truly understanding algorithm performance.
2. **Do** consider multiple performance metrics to understand your algorithm. You need not report all of these metrics; an exploratory phase helps guide final experiments.
3. **Do not** optimize over multiple performance metrics, simply to find which measure results in your algorithm having better performance than another. After the exploratory phase, decide what you want to measure. For final experiments reported in a paper, comparing your algorithm to others, you should consider additional/different environments from those used in the exploratory phase.
4. Offline and online returns are measuring very different things. Make the choice between these judiciously.
5. There are no clear-cut answers for defining and selecting evaluation metrics. You need to clearly justify your choices. A first criteria for the choice is whether you can convince yourself.
6. Most of our experiments should include behavioural metrics. We often test our agents in simple environments, where the goal is to understand our algorithms rather than obtain a solution in that MDP.