# Generalization Methodologies for Modern RL Algorithms

**Ruo Yu Tao***, **Derek Li***
Department of Computer Science
University of Alberta
Edmonton, AB T6G 2R3
{rtao3, xzli}@ualberta.ca

## Abstract

Generalization is an essential part of any intelligent agent. An intelligent agent should be able to generalize between similar scenarios. In reinforcement learning (RL) research, experiments with new algorithms are run to show performance on some desirable target distribution of conditions. But a mismatch occurs between this target distribution and the environment(s) that these experiments test on, since in most cases, agents are trained and evaluated on singular environments. This leads to the phenomenon of *environment overfitting*, where an agent or algorithm is designed to fit only that single environment, rather than generalizing to the target distribution they are actually looking to fit. In our work we look to correct this issue through introducing and developing *generalized methodologies*, where we evaluate algorithms based on samples for both tuning and testing environments from the same distribution. We first formalize a generalized version of the Mountain Car environment. We then motivate the use of generalized evaluation methodologies by showing the high variability of evaluating with singular environments. Finally, we use this methodology to test modern approaches to RL, including neural network function approximation and the components of the DQN algorithm. We show that neural network function approximation has better generalization capabilities across environments from the same target distribution. Code for this work is available at `https://github.com/taodav/generalization-rl`.

## 1 Introduction

One key aspect of intelligence is the ability for decision making agents to generalize (Chollet, 2019). Intelligent agents have an innate ability to learn from and adapt to new situations and experience - or to generalize from previous experience to new, novel experiences. Increasingly over the years, this has been a large topic of interest in reinforcement learning (RL) research.

The term "generalization" has different definitions at many scales in RL. When referring to generalization between states, we define generalization as a value function assigning similar states similar values (Sutton and Barto, 2018; Boyan and Moore, 1994). More recently, there has been an increased focus on generalization in a larger scale - the ability of agents and algorithms to generalize across a multitude of environments and hyperparameters. One approach to this generalization is to use a distribution of environments.

In the vast majority of RL experiments, the actual policy learned by the algorithm has little to no real-world use. The policy learnt by an algorithm on the Mountain Car environment doesn't actually help in any real-world scenarios. Instead, this learnt policy is meant to demonstrate the efficacy

*Equal contributions. Authors ordered alphabetically.

Preprint. Under review.

of a particular algorithm on a target distribution of conditions/environments that are desirable. For example, an algorithm developed to solve Montezuma's Revenge in the Arcade Learning Environment (Bellemare et al., 2013) most likely was not developed to only solve this video game, but meant to show that the algorithm is able to handle hard exploration problems with sparse rewards.

For RL experiments, how do we ensure that the algorithm is aligned to this target distribution? In the supervised learning setting, samples are taken from a target distribution and split into a training and testing set (Bishop, 2006) to measure the fit of a model to a target distribution. In most RL experiments, the "target distribution" is the distribution of rollouts/experiences on the same environment (with potentially some added stochasticity). This begs the further question - is this the right target distribution that we want to fit when developing RL algorithms?

This issue is encapsulated in the ability of an agent or algorithm to *generalize* across environments, or conversely, the issue of *overfitting* - where an algorithm or agent is only performant in a narrow range of settings and/or environments. Henderson et al. (2019) show that overfitting is particularly an issue for modern policy gradient methods that have unexpected results when changing factors such as neural network architecture, random seeds, implementation, etc. Zhang et al. (2018) and Farebrother et al. (2020) consider *individual agent* generalization, where an agent is trained on a set of training environments and are evaluated on a separate set of testing environments.

We instead focus on *algorithmic* generalization - where the designed algorithm itself generalizes well across a set of environments. An example of this is in Cobbe et al. (2019), where the authors show how "vanilla" versions of popular deep RL algorithms generalize better than their more specified counterparts in a single agent task transfer setting across different sampled environments.

In this work, we consider *generalized methodologies* in evaluating RL algorithms, where we instead focus on the ability of the algorithm itself to generalize instead of any single agent generalizing across multiple environments. Whiteson et al. (2011) introduces this concept as a way to avoid *environment overfitting* - where an algorithm is designed to only fit a single environment it was developed for, as opposed to the underlying distribution of environments the algorithm was meant to perform in.

In this work, we show that generalized evaluation methodologies are required to fairly evaluate the performance of an algorithm. We begin with some technical background behind RL and the introduction of generalized environments and evaluation methodologies behind these environments. We then empirically show issues with single environment evaluation methodologies - this is especially an issue when works use single environments as a representative of the performance of an algorithm on a more general set of problems and environments. To amend these issues, we develop the methodology behind evaluating with generalized environments. Specifically, we define a generalized version of the Mountain Car (Moore, 1990) environment, and develop an evaluation scheme for comparing algorithms with generalized environments. Finally, we use this methodology to perform an ablation study on the DQN (Mnih et al., 2015) algorithm and its components to test the efficacy of its component parts. Surprisingly, from these results we show that target networks are a hindrance to the DQN agent performance on our generalized Mountain Car environment. This serves as a demonstration of the efficacy of this evaluation scheme to more robustly evaluate modern algorithms.

## 2 Background

We begin with some background into both reinforcement learning and generalized methodologies for RL.

### 2.1 Formalizing reinforcement learning

To formalize our generalization methodologies, we use the Markov decision process (MDP) (Puterman, 1994) framework to describe decision-making problems in a given environment. In this framework, an agent interacts with this MDP by taking action $A_t \sim \mathcal{A}$ in state $S_t \sim \mathcal{S}$ over discrete time steps $t = 0, 1, \ldots$, where $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces respectively. The agent chooses actions given a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. With this action, the environment emits a reward $R_{t+1} \in \mathbb{R}$ and next state $S_{t+1} \sim \mathcal{S}$ according to the transition function $p(R_{t+1}, S_{t+1} \mid S_t, A_t)$.

In the control setting, our agent is seeking a policy to maximize the expected return:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

where $G_t$ is the discounted return defined as $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. We call this function the value function. We define an analogous action-value function that conditions on state and action:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a].$$

### 2.2 Function approximation in RL

We consider the case where our state space is too large to efficiently find the above functions for all states and instead rely on an approximation of these functions using learned parameters $\boldsymbol{w} \in \mathbb{R}^d$. We denote these new approximate functions as $\hat{v}(s, \boldsymbol{w})$ and $\hat{q}(s, a, \boldsymbol{w})$ for our value function and action-value functions respectively. In this work, to estimate an action-value function in the control setting, we use the $Sarsa(0)$ (Rummery and Niranjan, 1994) algorithm, with the update equation:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \delta_t \nabla_{\boldsymbol{w}_t} \hat{q}(S_t, A_t, \boldsymbol{w}_t)$$

where $\delta_t$ is the temporal difference error $\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)$. As for different options for our function approximator, two approaches that we consider are linear and neural network function approximation. For linear function approximation, our approximated action-value function is simply a linear combination of our state-action feature vector, $\boldsymbol{x}(S_t, A_t) \in \mathbb{R}^d$, and our weights:

$$\hat{q}(S_t, A_t, \boldsymbol{w}_t) \doteq \boldsymbol{x}(S_t, A_t)^\intercal \boldsymbol{w}_t$$

As for our neural network function approximator, we use a fully-connected neural network with non-linear activations for the action-value function approximation.

Besides this on-policy algorithm, we also consider the off-policy algorithm DQN (Mnih et al., 2015). Instead of the $Sarsa(0)$ temporal difference error, the DQN algorithm uses the Q-learning target for the TD error $\delta_t \doteq R_{t+1} + \gamma \max_{a \in \mathcal{A}} \hat{q}(S_{t+1}, a, \boldsymbol{w}_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)$. In addition, the DQN algorithm also uses neural network function approximation, as well as target networks and a replay buffer. A target network is a previous copy of the action-value approximator used for the target value in our temporal difference error. A replay buffer is a buffer of past experiences that the algorithm samples from at every update step to make batch updates.

### 2.3 Generalized methodologies

Development of recent RL algorithms has primarily followed the methodology of training and evaluating agents on a single environment to make generalized inferences. Currently, to ensure that agents do not overfit on an environment, various forms of stochasticity are added to the single environment (Machado et al., 2017) to ensure that agents do not take advantage of the determinism in some environments. While this has shown to help prevent some forms of overfitting, Whiteson et al. (2011) argue that when using only a single environment, the *algorithm* itself is overfit to the environment and lacks algorithmic generalization. We use their definition of *environment overfitting* to encapsulate this problem: through the process of algorithm development and hyperparameter tuning on a single environment, a learning algorithm is so customized to the environment that it performs poorly on other environments it was intended for. Continuing our example, this would be the case if the developed algorithm performed well on Montezuma's Revenge but poorly in other hard exploration tasks.

To combat overfitting, what may be a better idea would be for the algorithm to fit to a target distribution of environments instead of a single environment. To do this, Whiteson et al. (2011) evaluate agents on a specified *generalized environment* $\mathcal{G}$, defined as

$$\mathcal{G} \doteq (\Theta, \mu)$$

where $\Theta$ is the set of environments that we sample from and $\mu$ is the target distribution over $\Theta$ that we are interested in fitting. More specifically, Whiteson et al. (2011) define generalized environments by sampling environments from three specific generalizations: $\theta_i \in \Theta \doteq (\mathcal{N}, \mathcal{T}, \mathcal{I})$. First, at each environment step with action $A_t$, the change in environment state resulting from $A_t$ is perturbed based on a normal distribution with mean sampled from $\mathcal{N}$ and variance $0.05$. This is reminiscent of using sticky actions (Machado et al., 2017) in the ALE environments. Second, the observation that the agent receives is transformed based on a randomly sampled environment-specific transformation $\mathcal{T}$. Lastly, with equal probability, $\mathcal{I}$ specifies the probability in which we start with a fixed starting state or a random starting state.

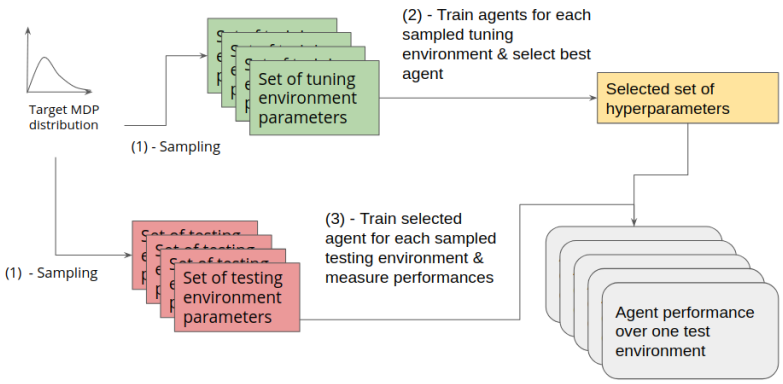



Figure 1: Generalized evaluation methodology for RL. (1) Sample two sets of environments according to $\mu$: a set of tuning environments and a set of testing environments. (2) For each instantiation of hyperparameters (over a set of hyperparameters to sweep) for the selected algorithm, train an agent with an instantiation of hyperparameters on each tuning environment and output some performance measure for each environment. Then, for each instantiation of hyperparameters, calculate some statistic over the set of tuning environments. Select the best hyperparameters based on comparing these statistics between instantiation of hyperparameters. (3) Using the selected hyperparameters, train agents on each of the test environments and return a set of performance measures for each test environment. Finally use these test performance measures to compare algorithms.

To evaluate an algorithm on a generalized environment, we use an evaluation methodology similar to that in supervised learning - we describe the approach in Figure 1.

## 3 Generalized Mountain Car

In order to compare the generalization capabilities of algorithms, we define a generalized version of the Mountain Car environment. We start with our definition of $\Theta$. In addition to the three generalizations $(\mathcal{N}, \mathcal{T}, \mathcal{I})$, we add an additional generalization $\mathcal{U}$ to define $\Theta \doteq (\mathcal{N}, \mathcal{T}, \mathcal{I}, \mathcal{U})$. $\mathcal{U}$ are parameters set for the environment update equation itself - for example in Mountain Car, the amplitude of the hill (while normally set to 1.0) could exist as a parameter sampled from $\mathcal{U}$. To make a generalized environment, we define the set of environments $\Theta$ and how we sample from this set $\mu$.

We first consider how the standard Mountain Car learning environment is defined. The action space of the environment is $A_t \in \{0, 1, 2\}$, where the 0 action is to accelerate backward, 1 is no-op and 2 is to accelerate forward. A standard Mountain Car state is defined as the vector $\boldsymbol{s}_t \doteq (p_t, v_t)^\intercal$, where $p_t \in [-1.2, 0.6]$ is the position (along the $x$-axis) of the car, and $v_t \in [-0.07, 0.07]$ is the velocity of the car. At every step, the position is updated with $p_{t+1} \doteq p_t + v_t$ and the velocity is updated with

$$v_{t+1} \doteq v_t + (A_t - 1) \times a + A\cos(3 \times p_t) \times (-g) \tag{1}$$

where $a$ is an acceleration parameter, $A$ is the amplitude of the hill and $g$ is the gravitational constant. In the standard Mountain Car environment these are normally set to $0.001$, $1.0$ and $0.0025$ respectively. We also have a period parameter that is set to a constant 3 in this environment.

Next we define the observation transformations, $\mathcal{T}$, for our generalized Mountain Car environment. The transformations $\mathcal{T}$ consists of sampling an offset and a normalization constant for both position and velocity (labelled as $p_{off}, v_{off}, p_{norm}, v_{norm}$). Applying a transformation $T \sim \mathcal{T}$ involves adding the offset and dividing by the normalization value to both position and velocity respectively. For observation and groundtruth state at timestep $t$, $\boldsymbol{o}_t$ and $\boldsymbol{s}_t$ respectively, we have:

$$\boldsymbol{o}_t \doteq T(\boldsymbol{s}_t) = \left( \frac{p_t + p_{off}}{p_{norm}}, \frac{v_t + v_{off}}{v_{norm}} \right).$$

Finally, we define our state update parameters, $\mathcal{U}$, set for Equation 1. In our work, we have $\mathcal{U} \doteq \{A\}$ - we sample the amplitude $A$ from some distribution over amplitudes. In this way, the height of the Mountain Car hill will vary from environment to environment - specifics of amplitude sampling are

described in Section 5. While we've decided to vary the amplitude for this generalized environment, the period and gravity parameters are also candidates for variation that fall out of the scope of this paper.

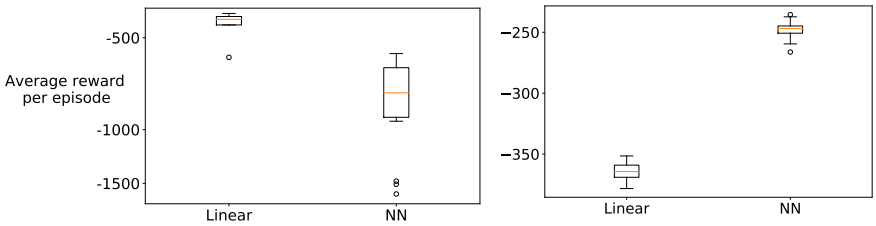
We add this additional generalization to introduce more variability in our generalized environment. Most generalizations from Whiteson et al. (2011) have already been introduced in similar forms in more recent RL literature (Machado et al., 2017) as measures to prevent overfitting. The addition of $\mathcal{U}$ allows us to vary the underlying MDP even more, requiring the tested algorithm to generalize further for good evaluation performance.

# 4 Single vs. multiple environments

We now describe our generalized methodology for comparing algorithms. We do so by comparing the generalization capabilities of algorithms. Before we describe our evaluation methodology, we first motivate it with an example.

## 4.1 Single environment evaluations

To understand why generalized environments are needed, we first consider two individual environments cherry-picked from sampling parameters as per Figure 3. In these two environments, we compare two variants of the $Sarsa(0)$ with tile coding algorithm - one with linear function approximation and one with neural network function approximation. Details of the algorithms, setup and hyperparameter choices are identical to Section 5.1 in that we choose hyperparamters based on the best total reward averaged over both episodes in 150k time steps and over 25 sampled tuning environments. In Figure 2, environment parameters in Figure 2a consist of $\{accel\_bias\_mean : 0.88, p\_offset : -0.37, v\_offset : -0.84, p\_noise\_divider : 5.13, v\_noise\_divider : 19.28, amplitude : 1.73\}$ and parameters in Figure 2b contain $\{accel\_bias\_mean : 0.85, p\_offset : -0.52, v\_offset : -0.91, p\_noise\_divider : 15.90, v\_noise\_divider : 11.23, amplitude : 1.73\}$. We compare the two algorithms by training agents on both environments and comparing average reward per episode over all episodes in 150k time steps.




(a) (b)

Figure 2: Numbers of steps to reach goal between two agents with linear and neural network function approximation, averaged over all completed episodes in two Mountain Car variants. Results are displayed in log scale. Left: linear function approximation outperforms neural network function approximation in environment a) Right: the neural network-based agent performs better on average in b). Results are averaged over 20 independent runs.

The result in Figure 2 demonstrates that small variations in the environment configuration can seriously impact conclusions we make about comparisons between RL agents. In this example, linear function approximation performs better than neural network function approximation in environment a), but ends up with much worse results in the environment b). Despite the fact that these results are aggregated over 20 independent runs, comparisons made in a single environment possess a large variance across environment variants and conclusions made based on these results are not reliable enough on their own.

### 4.2 Multiple environment evaluation

A better idea would be to evaluate our algorithms over multiple sampled environments from some target distribution that we are interested in. Exactly as in Figure 1, we follow Whiteson et al. (2011)'s evaluation methodology across some number of testing environments and perform evaluations for each environment. First, we tune hyperparameters on some number of sampled tuning environments and choose the set of hyperparameters that performs the best in terms of total rewards in an episode averaged over all episodes and over these tuning environments. After selecting the best hyperparameters for the given target distribution of environments, we use these hyperparameters to train agents on each of the sampled testing environments. For each of these test environments, we measure the average rewards in an episode over all episodes in a given number of steps. We describe the experiments run using this methodology in the following section.

## 5 Experiments

We now go in-depth into our experimental setup. We begin by describing how we sample from our generalized mountain car environment. In order to get samples of our environment, we sample our Mountain Car environment from separate uniform distributions for each parameter. For each of our tuning and test environments, we sample the following parameters as per Figure 3.

Amplitude: $A \sim U(0.75, 1.75)$
Car acceleration mean: $a' \sim U(0.8, 1.2)$
Position offset: $p_{off} \sim U(-1.0, 1.0)$
Velocity offset: $v_{off} \sim U(-1.0, 1.0)$
Position normalizer: $p_{norm} \sim U(5.0, 20.0)$
Velocity normalizer: $v_{norm} \sim U(5.0, 20.0)$.

Figure 3: Environment parameter distributions for our generalized mountain car environment. Note that $U(a, b)$ refers to the uniform distribution with the range $[a, b]$.

Altogether, we sample $25$ tuning environments and $100$ testing environments by sampling from the above distributions. We tune and test each algorithm on the same set of sampled environment parameters with the same seed. The discount rate $\gamma$ is 0.99. Note that we derive our car acceleration parameter $a$ with the following sampling and transformation at every step of the environment:

$$a \sim 0.001 \times |\mathcal{N}(a', 0.05)|$$

where $\mathcal{N}$ is a gaussian distribution. With these changes in both car acceleration and hill amplitude, one issue that arises is the solvability of a sampled Mountain Car environment. Due to the constraint on the maximum velocity of the car, and the fact that the acceleration generated by gravity is less than the acceleration generated by the car ($g < a$), at the maximum velocity there's a possibility that the hill is too steep for the maximum velocity to completely overcome the gravitational pull. This is especially true given the range of possible positions in the original Mountain Car environment. The entire environment is within the saddleback of two hills, but the minimum $x$-axis position allowed does not reach the top of the first peak, which gives less height for the car to accelerate down to reach the goal on the second peak. While this isn't an issue with the normal amplitude of $1.0$ in the original Mountain Car, this quickly becomes an issue with higher amplitudes. To remedy this, we also decrease our minimum position from $-1.2$ to $-1.5$ so that the car is able to reach the top of the first peak.

### 5.1 Function approximation in the control setting

Our first experiment attempts to examine the performance of $Sarsa(0)$ under linear and non-linear function approximations in the control setting. We first sweep over agent hyperparameter values $\alpha \in \{1.0, 0.5, 0.25, 0.125, 0.06125\}$, $num\_tilings \in \{4, 8, 16\}$, and $num\_tiles \in \{4, 8, 16, 32\}$ under 25 sampled tuning environments, where mean cumulative costs per episode are averaged

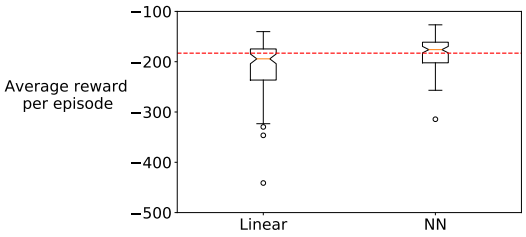



Figure 4: Numbers of steps to reach goal between two agents with linear (labeled *Linear*) and neural network (labeled *NN*) function approximation, averaged over all completed episodes of the generalized Mountain Car variants. Circles are outlier values below the minimum. The notched part of a box represents the 95% confidence interval of the median performance. The red dotted reference line refers to the bottom of the confidence interval for the neural network based algorithm, and is used to show that there is no overlap between the two 95% confidence intervals.

across all environments to select the best hyperparameter set. The two agent variants with chosen hyperparameters are then compared using 100 testing environments - in this setting, testing on each testing environment is considered as a "run". The agents follow an $\epsilon$-greedy policy with an epsilon of 0.1. The size of the hashtable used in tile-coding is 4096. Correspondingly, the fully-connected network used for non-linear function approximation consists of 2 layers of size 4096 and 2048 with the ReLU (Nair and Hinton, 2010) activation function and an output size equal to 3, the size of the discrete action space. We run a total of 150K timesteps for each environment, with no early termination of episodes.

We observe from the result in Figure 4 that there is greater variability in the performance of the linear $Sarsa(0)$ agent compared to the neural network-based agent. In addition, the comparisons between the median performance of the two agents indicate that neural network is likely to outperform linear function approximation most of the time since their 95% confidence intervals do not overlap but remain well separated by the red dotted reference line. We could conclude that under the same setup and feature representations (tile-coded features), a $Sarsa(0)$ agent is very likely to achieve higher cumulative rewards using a neural network than the standard linear method as function approximation, in a generalized Mountain Car environment. Compared to the conflicting results shown in the motivating example, this evaluation seems to be more robust to small variations in the configuration of an environment, which better represents the desired target distribution for the $Sarsa(0)$ agent. Therefore, our experimentation and evaluation strategy using generalized environments could help empirical results be more robust and reliable.

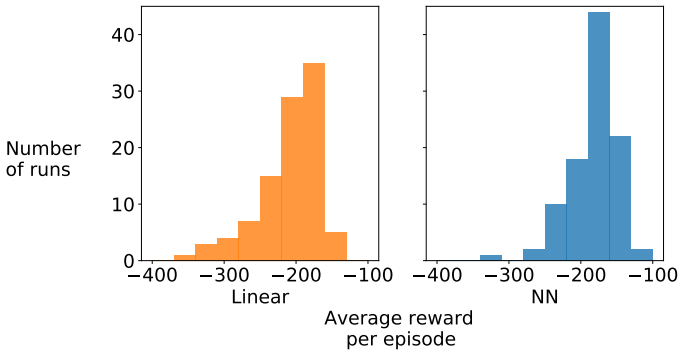



Figure 5: Histogram of agent's mean episodic return over 100 test environments. Both the linear and NN agent plots indicate skewed distributions of performance.

To further substantiate our argument, we analyzed the distribution of agent performance among 100 test environments. As expected, both linear and NN agents reveal a skewed distribution in their average episodic returns. This is reasonable since the agent could only do as well as an optimal policy in a Mountain Car environment. But, in the worst case, the sum of rewards is practically

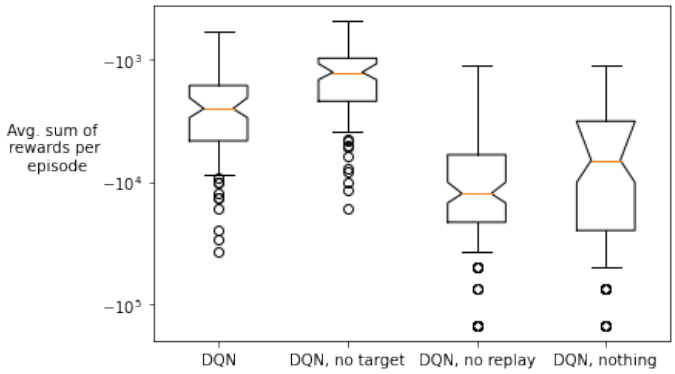



Figure 6: Results for variants of the DQN algorithm on the generalized mountain car environment. Each variant's best hyperparameters are tested for 150K steps on the 100 sampled test environments. Notches represent the 95% confidence interval around the median for each variant. Since there are no overlapping confidence intervals, we can conclude that for the generalized Mountain Car environment defined earlier, DQN without target networks is the most performant algorithm with a high degree of confidence.

unbounded below because of the stochasticity and difficulty of individual environments and the agent's inherent instability due to the stochastic learning dynamics. In addition, a constant learning rate $\alpha$ might have also contributed to this phenomenon. Finally, with the above factors, without an episode timestep limit an environment could theoretically continue on ad infinitum. As a result, the underlying distributional assumption made when estimating median performances is mildly violated. On the upside, the resulting distributions from the results do seem comparable since both agents' performances roughly follow the same family of distributions. Nonetheless, future work could help determine a statistical test that could better evaluate the results between two skewed distributions of performance metrics.

### 5.2 DQN ablation

The DQN (Mnih et al., 2015) algorithm proposed many algorithmic techniques that have been a catalyst for a lot of the recent progress in deep RL. While there has been some work on ablation studies on the different parts/additions to the algorithm (Hessel et al., 2017), these techniques have yet to come under the scrutiny of generalized environments.

In our work, we look to evaluate the efficacy of target networks and experience replay in the DQN algorithm on generalized environments. To do this, we use the same methodology as in Figure 1, and compare three variants of the Q-learning algorithm with tile coding and neural network function approximation - a variant with target networks, a variant with experience replay, a variant with both, and a baseline without either of the two algorithmic additions.

With the addition of these techniques comes the addition of hyperparameters. As before, we also sweep through a selection of these hyperparameters. For target networks, there is an additional hyperparameter for interval between target network updates - we sweep through the values $target\_interval \in \{4, 8, 16, 32\}$. For experience replay, we have an additional hyperparameter for replay buffer size - we sweep through the values $replay\_size \in \{1000, 25K, 150K\}$. We upper bound replay buffer size by 150K as that is the maximum number of steps for each agent in each environment.

For the rest of the experience parameters and settings, we use the same settings as Section 5.1. We compare average episodic rewards over all episodes for our test environments in Figure 6.

From the results, we conclude that for our generalized Mountain Car environment, we can say with high confidence that DQN without target networks is the most performant on this generalized environment. Also note that we use a log scale for the ticks on the y-axis. Based on these ablation results, we note that experience replay helps DQN performance significantly, whereas comparing DQN without a replay buffer with DQN without either a replay buffer or target networks we actually

see a slight drop in performance, although with higher variance when not using target networks when we don't use a replay buffer. We also observe the increased variance of our results as compared to the $Sarsa(0)$ with neural network function approximation and tile coding. It seems that either the off-policy nature of Q-learning and/or tile coding seem to dramatically affect the performance of the DQN algorithm as compared to our variant of $Sarsa(0)$.

While these results may have implications for the original DQN setup, we note that the DQN algorithm was developed using raw frame data in mind, as opposed to using the underlying state vector here with Mountain Car. For future work, running this DQN ablation on a generalized version the environments similar to the Atari 2600 games using frame data as proposed by Mnih et al. (2015) may yield more comparable insights into the DQN algorithm.

## 6 Conclusion

In this work, we investigated environment overfitting, a common phenomenon under the online evaluation setting. We showcased this problem with an experiment where we compared the performance of two $Sarsa(0)$ agents (one with linear function approximation and one with neural network function approximation) in two Mountain Car environments sampled from a distribution of Mountain Car environments. From this, we observed that single environment experiments can be misleading in comparing agent performances due to an underlying variability from testing on single environments. This presents an issue when works use single environments as a representative of the performance of an algorithm on a more general set of problems and environments. We then developed and empirically examined the generalized environment framework, where multiple sampled environments are used to produce evaluation metrics. Through these experiments, we showed its benefits in preventing environment overfitting, which produced less varied results. From these experimental results, we were able to show with statistical significance that for the $Sarsa(0)$ algorithm, neural network function approximation outperforms linear function approximation in generalized Mountain Car. Furthermore, using generalized environments, we were able to run an ablation study on the DQN algorithm with and without target networks and an experience replay buffer. From this experiment, we were able to show with statistical significance that the DQN algorithm without target networks performs better than the original DQN algorithm on the generalized Mountain Car environment defined previously.

Using techniques developed in this work, we believe that RL researchers could prevent environment overfitting and enjoy more success in attaining statistically significant comparisons through adopting generalized methodologies. This also opens up new opportunities to revisit established RL algorithms such as deep policy gradient methods under this new framework. This methodology of fitting to a target distribution of environments rather than a single environment also enables a more robust test bed for evaluating algorithms on specific sub-problems in RL. For example, if an RL researcher wanted to test the exploration capability of their algorithm, they could develop a target distribution of environments specifically for exploration problems and test their algorithm on this generalized environment, showing the exploration capabilities on a range of environments rather than just a single environment. Through development of algorithms on generalized environments, we hope that RL algorithms developed this way will be more robust to hyperparameter and environment variations.

## Acknowledgements

We would like to thank Brian Tanner for the extensive help he provided in the implementation of the generalized environments and methodologies used in this work.

## References


Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Boyan, J. A. and Moore, A. W. (1994). Generalization in reinforcement learning: Safely approximating the value function. In *Proceedings of the 7th International Conference on Neural Information Processing Systems*, NIPS'94, page 369–376, Cambridge, MA, USA. MIT Press.

Chollet, F. (2019). On the measure of intelligence.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2019). Quantifying generalization in reinforcement learning.

Farebrother, J., Machado, M. C., and Bowling, M. (2020). Generalization and regularization in dqn.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2019). Deep reinforcement learning that matters.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. (2017). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–33.

Moore, A. (1990). Efficient memory-based learning for robot control.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA. Omnipress.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., USA, 1st edition.

Rummery, G. and Niranjan, M. (1994). On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Whiteson, S., Tanner, B., Taylor, M. E., and Stone, P. (2011). Protecting against evaluation overfitting in empirical reinforcement learning. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 120–127.

Zhang, C., Vinyals, O., Munos, R., and Bengio, S. (2018). A study on overfitting in deep reinforcement learning.