Towards Deep Reinforcement Learning for Continuing Control Tasks with Average Reward

Kenneth Tjhia[1] **Jeremy Costello**[2] **Vadim Avkhimenia**[2] **Kostyantyn Guzhva**[1]
{tjhia,jeremy1,avkhimen,,guzhva}@ualberta.ca

[1]Department of Computing Science
University of Alberta

[2]Department of Electrical and Computer Engineering
University of Alberta

Abstract

We compare off-policy, model-free, value iteration based average reward reinforcement learning (RL) control algorithms with multi-layer perceptron (MLP) function approximation (FA) on two continuing control tasks. The compared algorithms are Differential Q-learning [1], RVI Q-learning [2], and R-learning [3]. Algorithms are evaluated on final attained average reward rates, sample efficiencies, and hyperparameter sensitivity. This work is the first to extend Differential Q-learning and RVI Q-learning to the function approximation setting with MLPs, and the first to compare all three algorithms in this setting. R-learning had the best sample efficiency, Differential Q-learning achieved the highest final attained average reward rate, and all three algorithms had relatively similar hyperparameter sensitivities.

1 Introduction

Most continuing RL algorithms use temporal discounting to ensure a potentially infinite sum of values is finite. Discounting requires the tuning of a discount factor, γ. Setting the discount factor too low results in learning a non-optimal policy, while setting it too high results in slower learning. It is difficult to determine the optimal discount factor a priori, so many algorithm implementations are non-optimal in terms of policy or learning speed.

R-Learning [3] remedies this by using average-reward instead of discounting, removing the difficulty of choosing an optimal discount factor. R-learning is a control algorithm; it aims to find a policy which maximizes reward. Following R-Learning, several other average-reward control algorithms have been introduced, including a modification of R-learning [4], which we suggest to study in future research, RVI Q-learning [2], and Differential Q-learning [1].

The average-reward RL framework objective is to select policies that generate the highest expected payoff per time-step [5]. An empirical study of R-Learning was performed on a stochastic grid world containing one-way membranes and a simulated robot environment. R-Learning outperformed Q-Learning provided sufficient exploration. R-Learning has two parameters: $0 \leq \alpha \leq 1$, the action-value update step-size, and $0 \leq \beta \leq 1$, the step-size for updating an estimate of the assumed-to-exist optimal long run average-reward rate, r^*. A sensitivity analysis on α and β was performed for R-Learning, which found that initializing α moderately large, β small, and decaying α at a slow rate resulted in higher values of cumulative reward. It was also shown that R-learning reacts more acutely to exploration than discounted algorithms and can fall into limit cycles.

Differential Q-Learning is the first average-reward control algorithm not requiring reference states and proven-convergent in the tabular setting [1]. The performance of Differential Q-Learning and RVI Q-Learning was compared on the Access-Control Queuing task [6] in the tabular setting. Both algorithms converged, however Differential Q-Learning demonstrated greater stability over the chosen range of hyperparameters.

Many RL environments are too large for tabular representation of state-action pairs. Function approximation can be used to provide generalization across the state-action space. Many average-reward algorithms exist; some are proven to converge in the tabular setting, and others perform well in practice but are yet to be proven-convergent. Despite a lack of convergence guarantee for Q-learning under non-linear function approximation [7], Deep Q-Networks (DQN) [8] achieved state-of-the-art performance on six episodic Atari environments. This shows that a formal proof of convergence under function approximation is not a strict necessity for real-world performance.

R-Learning has been implemented in the function approximation setting using an encoder-decoder convolutional neural network architecture [9], but to our knowledge Differential Q-Learning and RVI Q-Learning have not been extended to the function approximation setting with MLPs, and no comparison has been performed between them in this setting

Accordingly, we find it reasonable to extend these tabular algorithms to incorporate function approximation, even when under such conditions we lack a guarantee of convergence. We hope a comparison of average-reward algorithms in continuing tasks will allow future RL designers to make more informed decisions on which algorithm to use for different classes of tasks. This paper provides a background on each average-reward algorithm, our extensions to the FA setting using MLPs, and performance comparisons between all three algorithms over a range of hyperparameterson on two continuing environments from the PyGame Learning Environment [10].

2 Background

2.1 Average Reward MDPs

A discrete-time, infinite-horizon, finite Markov decision process, henceforth referred to as an MDP, is a tuple $\mathcal{M} \doteq (\mathcal{S}, \mathcal{A}, p, r, \mu_0)$, where $\mathcal{S}$ is a finite (non-empty) set of states, $\mathcal{A}$ is a finite (non-empty) set of actions, $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ gives the probability of transitioning to state s' when action a is taken in state s, written $p(s'|s, a)$, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ gives the *expected* reward when action a is taken in state s, and μ_0 is the initial state distribution. Here we only consider the case where actions are chosen according to a stationary, deterministic policy, a function $\pi : \mathcal{S} \to \mathcal{A}$. Such a policy, together with an MDP, generates a sequence $(S_t, A_t, S_{t+1}, R_{t+1})_{t=0}^{\infty}$ as follows:

1. Sample the initial state s_0 from μ_0.
2. Repeat for $t = 0, 1, 2 \ldots$
   (a) Take action $a_t = \pi(s_t)$.
   (b) Transition to state s_{t+1} sampled $p(\cdot|s_t, a_t)$ and receive reward $r_{t+1} = r(s_t, a_t)$.

Note that the subsequence $(S_t)_{t=0}^{\infty}$ is a time homogeneous Markov chain with transition probabilities

$$\mathbb{P}\left(S_{t+1} = s' | S_t = s\right) = p(s'|s, \pi(s))$$

for $s, s' \in \mathcal{S}$. If, for some MDP, every stationary policy induces a Markov chain containing exactly one recurrent class and a single, possibly empty, set of transient states, we say the MDP is unichain. We restrict all further discussion to unichain MDPs. Fix some unichain MDP. The long term reward rate (also known as average reward, or gain, in the literature) of state $s \in \mathcal{S}$ with respect to policy π is

$$r_\pi(s) \doteq \lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} \mathbb{E}\left[R_t | S_0 = s, \pi\right]$$

which exists and is independent of s [11], hence we can write r_π for the average reward of policy π. Note that, due to the unichain assumption, the contribution of reward from transient states to this average vanishes in the limit $n \to \infty$. This objective does not discriminate between policies which, in expectation, accumulate different amounts of reward, or require more transitions, before settling in the recurrent class. The differential value (also known as average-adjusted value, relative value, or bias in the literature) of state $s \in \mathcal{S}$ with respect to a policy π is defined to be

$$v_\pi(s) \doteq \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \sum_{t=1}^{k} \mathbb{E}\left[R_t - r_\pi | S_0 = s, \pi\right]$$

which also exists [11]. Let $r_{\pi^*} = \max_{\pi \in \Pi} r_\pi$, where Π is the set of all stationary policies. Then the corresponding state-value Bellman optimality equations are

$$v(s) = \max_{a \in \mathcal{A}} \left\{ r(s,a) - r_{\pi^*} + \sum_{s' \in \mathcal{S}} p(s'|s,a) v(s') \right\}$$

for $s \in \mathcal{S}$. This system of equations has a set of solutions $\{v_{\pi^*}(s) + c | v_{\pi^*}(s) = \max_{\pi \in \Pi} v_\pi(s), c \in \mathbb{R}\}$ for $v(s)$ [11]. From this it follows that the corresponding action-value Bellman optimality equation

$$q(s,a) = r(s,a) - r_{\pi^*} + \sum_{s' \in \mathcal{S}} p(s'|s,a) \max_{a' \in \mathcal{A}} q(s',a') \tag{1}$$

has a set of solutions $\{q_{\pi^*}(s,a) + c | c \in \mathbb{R}\}$, where

$$q_{\pi^*}(s,a) = r(s,a) - r_{\pi^*} + \sum_{s' \in \mathcal{S}} p(s'|s,a) v_{\pi^*}(s')$$

for $s \in \mathcal{S}$ and $a \in \mathcal{A}$. A solution to an MDP under the average-reward criteria is a policy whose corresponding action-value function satisfies Equation 1. Accordingly, the algorithms we compare here attempt to approximate r_{π^*} and a solution to $q(s,a)$. Further, the policy update steps of these control algorithms only make use of the differences between action-value pairs, so the fact that the $q(s,a)$ solutions to Equation 1 are only unique up to a constant offset causes no issue.

2.2 Algorithms

We next give an overview of the three algorithms and describe them as they were originally presented, for the tabular setting. Section 3.1 describes our extension of these using MLP function approximators, and appendix A provides the pseudocode of our implementations.

In the tabular setting, R-learning has not been proven convergent, but if it converges then it does so to a solution to Equation 1 [5]. It has been demonstrated empirically, however, that R-Learning may diverge even for unichain MDPs, suggesting that no proof of convergence exists [5]. The pseudocode is shown in Algorithm 1.

Algorithm 1: R-Learning

Input: The policy π to be used
Algorithm Parameters: Step sizes $\alpha, \eta > 0$
Initialize action-value value function $Q(s,a)$ arbitrarily $\forall s, a$
Initialize average reward estimate $\bar{R}$ arbitrarily
Obtain initial state S
while *still time to train* **do**
  Select action A according to π and S
  Take action A, observe reward R and next state S'
  $y = R - \bar{R} + \max_a Q(S',a) - Q(S,A)$
  $Q(S,A) = Q(S,A) + \alpha y$
  if A *was a greedy action* **then**
    $\bar{R} = \bar{R} + \eta \alpha y$
  end
  $S = S'$
end

In the tabular setting, RVI Q-Learning has been proven to converge to a solution to Equation 1 under some mild assumptions [2]. The algorithm incorporates a function f of the action-value function Q; one of the aforementioned assumptions is that this f satisfies the following properties:

1. f is Lipschitz.
2. $f(\mathbf{e}) = 1$ where $\mathbf{e}$ is a $|\mathcal{S}| \times |\mathcal{A}|$ dimensional vector of 1's.
3. $f(\mathbf{x} + c\mathbf{e}) = f(\mathbf{x}) + c$ for all $c \in \mathbb{R}$.

One choice of f that satisfies these assumptions is $f(Q) = Q(s_0, a_0)$, the action-value function evaluated at a fixed *reference* state-action pair. With this choice of f, it has been shown empirically that rate of convergence depends strongly on the chosen reference state-action pair [2]. Performance was found to generally be better when the reference state-action pair is visited frequently under the optimal policy, and the algorithm may diverge when the pair is transient under this policy. The pseudocode is shown in Algorithm 2.

Algorithm 2: RVI Q-Learning

Input: The policy π to be used
Input: The function $f(Q)$ (for e.g. $f(Q) = Q(s_0, a_0)$ for chosen s_0, a_0
Algorithm Parameters: Step sizes $\alpha, \eta > 0$
Initialize action-value value function $Q(s, a)$ arbitrarily $\forall s, a$
Obtain initial state S
while *still time to train* **do**
  Select action A according to π and S
  Take action A, observe reward R and next state S'
  $y = R - f(Q) + \max_a Q(S', a) - Q(S, A)$
  $Q(S, A) = Q(S, A) + \alpha y$
  $S = S'$
end

Differential Q-learning is the first model-free, off-policy control algorithm that does not require one to choose a function f, or reference states, as in RVI Q-learning, and is proven-convergent for the tabular setting [1]. Similar to R-Learning, it maintains an estimate of the average reward $\bar{R}$, however, it updates this estimate on every time step regardless of whether the action was greedy. The pseudocode is shown in Algorithm 3.

Algorithm 3: Differential Q-Learning

Input: The policy π to be used
Algorithm Parameters: Step sizes $\alpha, \eta > 0$
Initialize action-value value function $Q(s, a)$ arbitrarily $\forall s, a$
Initialize average reward estimate $\bar{R}$ arbitrarily
Obtain initial state S
while *still time to train* **do**
  Select action A according to π and S
  Take action A, observe reward R and next state S'
  $y = R - \bar{R} + \max_a Q(S', a) - Q(S, A)$
  $Q(S, A) = Q(S, A) + \alpha y$
  $\bar{R} = \bar{R} + \eta \alpha y$
  $S = S'$
end

2.3 Environments

We next describe the environments that we run our experiments in. Both are from the PyGame-Learning Environment (PLE) [10], a Python library that implements several Atari 2600 games for the purpose of evaluating RL algorithms.

The default Catcher PLE environment (Figure 1) an episodic task. The agent controls a rectangular paddle at the bottom of a two dimensional screen, its goal is to move this paddle to catch fruits that fall from the top of the screen with a constant velocity. There is only one fruit on the screen at any time. The possible actions are moving the paddle left, or moving the paddle right, and there is some acceleration before the paddle reaches a maximum speed in either direction. The agent gets a reward of $+1$ for successfully catching a fruit and a reward of -1 for missing a fruit. The original environment terminates when the agent fails to catch some prescribed number of fruits; we removed

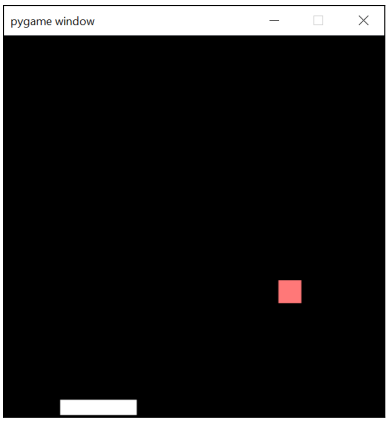

Figure 1: Catcher environment

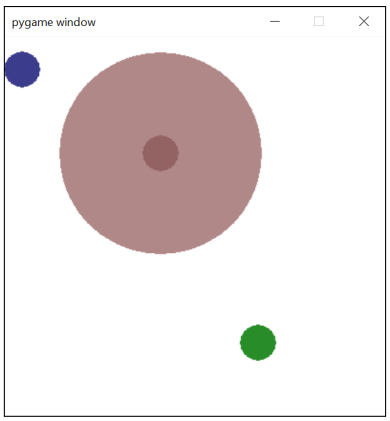

Figure 2: Puckworld environment

this condition to turn it into a continuing task. In each time-step the agent observes the x, y-position of the paddle, its velocity, and the x, y-position of the fruit. All positional and velocity values are from a continuous range. Figure 1 shows a snapshot of the environment.

The PuckWorld PLE environment (Figure 2) is a continuing task in which the agent controls a blue dot. The agent's objective is to navigate towards a green dot while avoiding a larger red circle of constant radius. The red circle slowly follows the agent, while the green dot randomly moves around on the screen. Possible actions are moving the blue dot up, down, left, or right. Every time-step, the agent receives a negative reward proportional to distance from the green dot. If the agent is within the red circle, it receives an additional negative reward proportional to its distance form the edge of the circle. The observation consists of the x, y-positions of the blue dot, green dot, and center of the red circle, and the x, y-velocity of the blue dot. All positional and velocity values are from a continuous range. Figure 2 shows a snapshot of the environment.

3 Experiments

3.1 Extension to FA

We now describe how we extend the three algorithms —R-learning, RVI Q-learning, and Differential Q-learning —to the FA setting, beginning with details common to all of our implementations, and then discuss some algorithm specific implementation details. Pseudocode is provided in Appendix A, and our implementations are available at https://github.com/hexken/average-reward-benchmarking.

We use MLP function approximators for the action-value function closely following Deep Q-Learning with Experience Replay [8]. All implementations have two hidden layers with $\lceil (2/3) \cdot \text{num\_inputs} + \text{num\_ outputs} \rceil$ units per hidden layer, a rule of thumb suggested in [12]. We take as input the raw observations provided by provided by PLE; our network for Catcher has 5 hidden units per layer and our network for PuckWorld has 10 hidden units per layer. The output layers have a single node representing each action.

We use a vanilla experience replay buffer to mitigate problems caused by the temporal correlations of consecutive observations received from the environment [8]. We set the buffer capacity equal to the number of training steps and perform updates on randomly sampled batches of size $n_b = 32$ once the buffer has at least n_b experiences.

We use a vanilla target network to provide a more stable training target [8]. We update the target network every 1000 steps.

We use the Huber loss function [13] with $\delta = 1$. Using Huber rather than mean squared error reduces the chance of encountering exploding gradients [14].

We use an Adam optimizer [15] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1.5 \times 10^{-4}$ (ϵ the Adam parameter) as in [16]. We clamp each component of our gradient to $[-1, 1]$ to prevent exploding gradients [17]. The α parameter for Adam, i.e. the learning rate for our Q network update and a

factor in the step-size for our average reward update (R-Learning and Differential Q-Learning) is one of the hyperparameters we sweep over.

All algorithms use an ϵ-greedy behaviour policy which chooses a random action with probability $0 \leq \epsilon \leq 1$, and chooses the action maximizing our current estimate of $Q(s, a; \mathbf{w})$ with probability $1 - \epsilon$. To encourage initial exploration we initialize ϵ to 1, perform 1000 steps at this setting, then linearly decay it over the next 4000 steps to 0.1. Note that this ϵ is separate from the parameter for Adam.

For RVI Q-Learning and Differential Q-Learning, which both maintain an estimate of the average reward, we use the mean reward of the sampled batch to update this estimate. See Appendix A for details.

For R-Learning, the specification for the tabular setting dictates that the average reward estimate is only updated during time-steps where the greedy action is taken. Accordingly, we remove non-greedy experiences from the sampled batch before averaging the rewards.

For RVI Q-Learning, we choose as an f function the mean action-value evaluation of the first n_r state-action pairs visited. n_r is one of the hyperparameters we sweep over. We only perform updates once at least $\max n_r, n_b$ experiences have been collected.

3.2 Methodology

Three algorithms —R-learning, RVI Q-learning, and Differential Q-learning —each extended to the FA setting, are compared on two environments: PuckWorld (Figure 2) and Catcher (Figure 1), both from PLE [10]. Algorithms are compared on the basis of their average reward over all time-steps, average reward over the last 5000 time-steps, and hyperparameter sensitivity. This is, to our knowledge, the first comparison of this kind.

Hyperparameter sweeps for each algorithm-environment combination are performed over a grid. The best-performing hyperparameter combination for each algorithm and environment are compared. Hyperparameter sensitivity is investigated by comparing the final average reward rate for each algorithm-environment combination. A starting point for hyperparameter ranges was based on work in the tabular setting [1, 5], although these ranges were modified for the function approximation setting with MLPs.

Catcher and PuckWorld are interesting environments on which to perform this comparison due to continuous state spaces, necessitating the use of function approximation. We believe these environments will give realistic performance representations of each average reward algorithm. Despite this, a potential limitation of this investigation is that both environments are small and use object representations, unlike the large Atari environments and pixel representations that DQNs were tested on [8]. Due to this, results may not represent how well algorithms will perform on larger environments such as MuJoCo.

3.2.1 Hyperparameter Sweeps

Two hyperparameters are swept over for each algorithm. All three algorithms sweep over α, the learning rate for the Adam optimizer, and a factor in the step-size for updating the average reward estimate for R-Learning and Differential Q-Learning. The values of α sweep over are $10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$. This range was chosen based on preliminary testing on the PuckWorld environment, during which performance deteriorated when the learning rate was set too high (10^{-1}) or too low (10^{-4}).

For R-learning and Differential Q-learning, the second parameter swept over was η, the second factor in the average reward update step-size. The values of η swept over are $10^{1}, 10^{0}, 10^{-1}, 10^{-2}$. These values were chosen with a similar method to the α sweep values.

For RVI Q-learning we sweep over n_r, the number of initially visited state-action pairs that become our set of reference state-actions that are averaged over for our f function. This is analogous to the f function which evaluates the action-value of a single reference state-action pair in the tabular setting. We chose this because the continuous state spaces in these environments made it difficult to select a single reasonable state-action pair, which may be infrequently visited. The values of n_r we sweep over are $16, 32, 64, 128$.

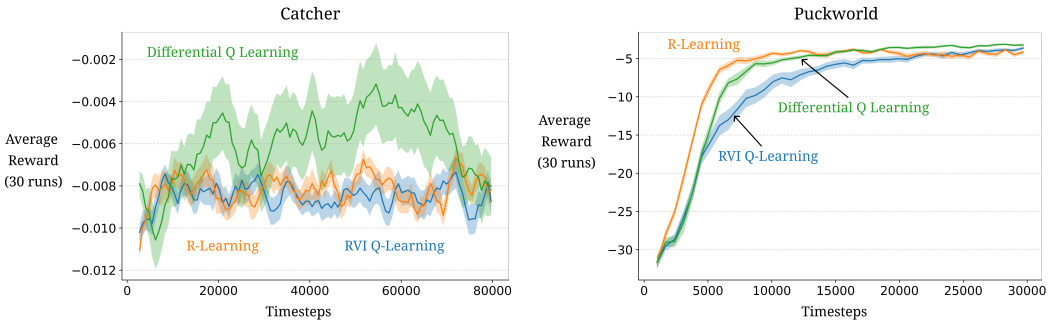

Figure 3: Learning curves of each environment and algorithm combination using the best hyperparameter configuration, as determined by the greatest average reward over all time steps.

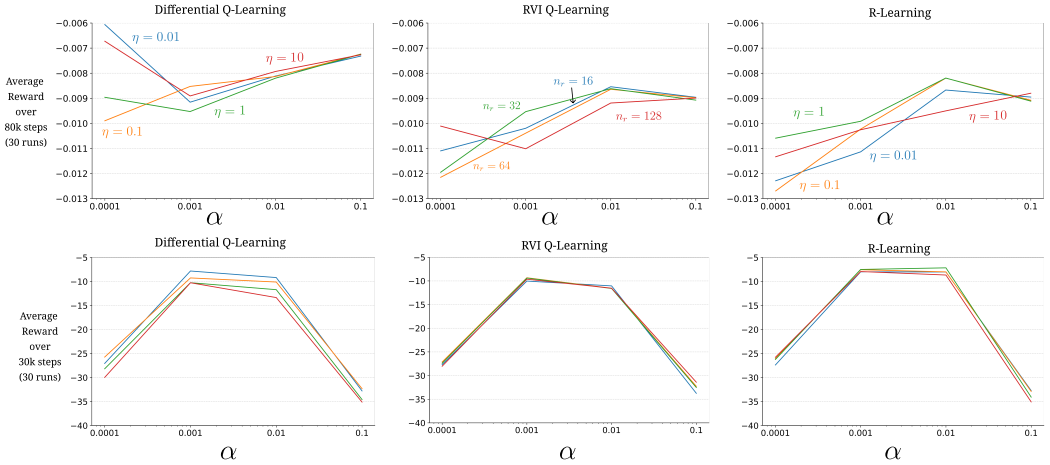

Figure 4: Sensitivity Plots. The top row is for the Catcher environment, the bottom is for the PuckWorld environment. The colours representing different hyper-parameter values are consistent across plots.

Each algorithm was run 30 times for each hyperparameter combination. For PuckWorld each run is 3×10^4 steps, for Catcher each run is 8×10^4 steps.

4 Empirical Results

For each environment, we determined a best hyperparameter configuration by averaging the reward over all time steps, over all runs. The learning curves corresponding to these configurations are shown in Figure 3. In Figure 4 we plot the reward averaged over all time steps for each environment, algorithm, and hyperparameter configuration.

None of the algorithms perform particularly well on Catcher, although Differential Q-learning seems to have the most potential. This may be due to sparse rewards or insufficient exploration. On average, there are 36 time-steps between non-zero rewards in Catcher, while non-zero rewards are observed each time-step in PuckWorld. This should be explored further. Due to current poor performance on Catcher, we do not believe any inferences can be made on algorithm performance and hyperparameter sensitivity from the data obtained in the Catcher experiments. Henceforth, all algorithm performance and hyperparameter sensitivity comparisons will be made based exclusively on data obtained in the PuckWorld experiments.

R-learning appears to be the most sample efficient algorithm, although this could be affected by epsilon decay, something which should be further investigated. R-learning's sample efficiency is followed by Differential Q-learning, and trailed quite substantially by RVI Q-learning. The only difference between R-Learning and Differential Q-learning is that in R-Learning we remove non-greedy actions from the sampled batches when updating the average reward estimate. Differential

Q-learning eventually overtakes R-learning in terms of average reward over the last few thousand steps, showing that its update rule (in this case) is better in the long run. We believe this warrants further investigation.

Differential Q-learning achieves the highest reward averaged over all time-steps, although the other two algorithms are fairly close and RVI Q-learning appears to still be improving when training was stopped. Average reward over all time-steps is an important metric for comparing algorithm performance as it incorporates both final attained average reward rate and also the rate of convergence (when it occurs). We believe further comparisons should be performed on more environments to decisively determine the best-performing algorithm in terms of this metric.

We evaluate the algorithms by comparing the rewards averaged over all time-steps (averaged over all 30 runs) by the their rewards averaged over the last 5000 time-steps. For each algorithm and environment, a single (possibly different) hyperparameter configuration maximized both quantities. All significance tests are performed at the 0.01 significance level. For reward averaged over all time-steps, we use the Levene test [18] and find that the sample variances are different between the algorithms. Therefore we use the Welch's ANOVA test [19] the null hypothesis that all algorithms have the same average reward, and find there is sufficient evidence to reject the null hypothesis with p-value of 0.00031.

Next we use the Games-Howell test [20] to perform pairwise tests of the algorithms. We find that the difference between R-learning and RVI Q-learning is statistically significant and R-learning is better with p-value of 0.001, but no other pair has a statistically significant difference.

When we compare the rewards averaged over the last 5000 time steps we use a similar procedure. A Levene test tells us that the variances are equal, so we use ANOVA [21] rather than Welch's ANOVA and find that the average rewards are different with a p-value of 0.000539. We then use pairwise Tukey-HSD [22] tests instead of Games-Howell (because of the unequal variances). We find that there is a statistically significant difference between Differential Q-learning and R-learning, where Differential Q-learning is better with p-value of 0.001, but there is no significant difference between other pairs. It should be noted that although these differences are statistically significant, the difference in magnitude of the average reward achieved by these algorithms is relatively small.

All algorithms exhibit similar hyperparameter sensitivity. Adam optimizer learning rates (α) of 10^{-2} and 10^{-3} perform well across algorithms, but performance deteriorates for learning rates of 10^{-1} and 10^{-4}. Learning rate of 10^{-3} slightly outperforms 10^{-2} for Differential Q-learning and RVI Q-learning, while there isn't a clear difference in performance for R-learning. RVI Q-learning does not exhibit much sensitivity to n_r and R-learning does not exhibit much sensitivity to the average reward step-size modifier, η. Differential Q-learning exhibits more sensitivity to η, with lower values of η resulting in slightly higher average reward.

In the tabular setting, the performance of RVI Q-learning is highly dependent on how often the reference state is visited [1]. The generalization of MLPs in function approximation allows the MLP to adequately estimate average reward at the reference states without directly visiting them. We believe this is why RVI Q-learning performs at a similar level to R-learning and Differential Q-learning in this experiment, which was unexpected.

5 Conclusion

Three average reward algorithms were compared in the function approximation setting —Differential Q-Learning, R-Learning, and RVI Q-Learning algorithms —on two continuing environments: Catcher and Puckworld. We judged algorithm performance exclusively from the Puckworld environment because none of the algorithms converged on Catcher and all of the agents exhibited similar random-looking behavior. R-Learning was found to be the most sample efficient, followed by Differential Q-Learning, however in the long run the average reward obtained using Differential Q-Learning was higher, likely due to the more frequent update of average reward. All algorithms exhibited similar hyperparameter sensitivity, with α of 10^{-3} preferable for Differential Q-Learning and RVI Q-Learning. Differential Q-Learning achieved the largest average reward rate but other algorithms were close and more investigation is required here. RVI Q-Learning achieved a similar level of average reward compared with other algorithms because generalization of MLPs reduces the effect of less-visited states on the performance of the algorithm.

6 Future Work

We have identified a few areas for future work. Choosing the reference states for RVI Q-learning is an important part of the algorithm. How these are chosen under MLP function approximation, whether these choices can be updated as the algorithm runs, and the applicability of RVI Q-learning's convergence criteria should be investigated. Whether it is better to update the average reward update for R-learning and Differential Q-learning from the experience replay buffer or directly from experience, and how the difference in when each of these algorithms updates average reward affects performance should be investigated. More hyperparameter choices and different MLP architectures should be investigated.

Poor performance on the Catcher environment should be investigated. Methods which could improve performance are prioritized replay [23], entropy regularization [24], and slowing epsilon decay. We believe the sparse rewards on Catcher are causing much of the experience replay buffer to quickly become useless for learning, so prioritizing samples from the buffer based on how much they can teach the agent may improve performance. We also believe the agent is insufficiently exploring, which could be rectified with entropy regularization or by slowing the epsilon decay. Entropy regularization discourages action predictability by adding an entropy term to the loss function, resulting in more exploration [25]. This would require the MLP to predict softmax action probabilities rather than action values, and these action probabilities would be used for agent action selection rather than ϵ-greedy. Slowing epsilon decay will cause the agent to take more random actions near the start of training, increasing early exploration.

Further testing should be performed on these algorithms. In Figure 3, RVI Q-learning still appears to be improving in the PuckWorld environment when training was halted. The effect of epsilon decay on sample efficiency should be investigated, since the R-learning curve for PuckWorld in Figure 3 looks like it could just be following the epsilon decay schedule. These experiments should be run again for more time steps. These algorithms should also eventually be tested on larger environments, where pixel representations could be used instead of object representations. This would require convolutional neural networks to be used rather than MLPs.

References

[1] Yi Wan, Abhishek Naik, and Richard S Sutton. Learning and planning in average-reward markov decision processes. *arXiv preprint arXiv:2006.16318*, 2020.

[2] Jinane Abounadi, Dimitrib Bertsekas, and Vivek S Borkar. Learning algorithms for markov decision processes with average cost. *SIAM Journal on Control and Optimization*, 40(3): 681–698, 2001.

[3] Anton Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the tenth international conference on machine learning*, volume 298, pages 298–305, 1993.

[4] Satinder P Singh. Reinforcement learning algorithms for average-payoff markovian decision processes. In *AAAI*, volume 94, pages 700–705, 1994.

[5] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine learning*, 22(1-3):159–195, 1996.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[7] John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-difference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[9] Rishi Shah, Yuqian Jiang, Justin Hart, and Peter Stone. Deep r-learning for continual area sweeping, 2020.

[10] Norman Tasfi. Pygame learning environment. `https://github.com/ntasfi/PyGame-Learning-Environment`, 2016.

[11] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[12] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[13] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer, 1992.

[14] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[17] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[18] Howard Levene. Robust tests for equality of variances. In *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, pages 278–292. 1960.

[19] Bernard L Welch. The generalization of student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[20] Paul A Games and John F Howell. Pairwise multiple comparison procedures with unequal n's and/or variances: a monte carlo study. *Journal of Educational Statistics*, 1(2):113–125, 1976.

[21] Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics*, pages 66–70. Springer, 1992.

[22] John W. Tukey. Comparing individual means in the analysis of variance. *Biometrics*, 5(2):99–114, 1949. ISSN 0006341X, 15410420. URL `http://www.jstor.org/stable/3001913`.

[23] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

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

[25] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

A Algorithms

Algorithm 4: DQN with Differential Q-Learning

Input: The policy π to be used
Input: Differentiable loss function $loss\_fn$
Algorithm Parameters: Step sizes $\alpha, \eta > 0$
Algorithm Parameters: Batch size n_b
Algorithm Parameters: Number of steps between target network updates τ
Algorithm Parameters: Number of steps to train for T

Initialize action-value network Q with random weights $\mathbf{w}$
Initialize target network weights $\mathbf{w}^- = \mathbf{w}$
Initialize empty experience replay buffer $\mathcal{B}$
Initialize average reward estimate $\bar{R} = 0$
Obtain initial state S_1
for $t1$ **to** T **do**
    Select action A_t according to π
    Take action A_t, observe R_{t+1} and S_{t+1}
    Store experience $(S_t, A_t, R_{t+1}, S_{t+1})$ in $\mathcal{B}$
    if *$\mathcal{B}$ has at least n_b experiences* **then**
        Let $(S_{i_j}, A_{i_j}, R_{i_{j+1}}, S_{i_{j+1}})_{j=1}^{n_b}$ be a random sample of n_b experience from $\mathcal{B}$
        for $j = 1$ **to** n_b **do**
            $y_j = R_{i_{j+1}} - \bar{R} + \max_a Q(S_{i_{j+1}}, a; \mathbf{w}^-)$
            $\ell_j = loss\_fn(y_j, Q(S_{i_j}, A_{i_j}; \mathbf{w})$
        end
        $\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} \left((n_b)^{-1} \sum_{j=1}^{n_b} \ell_j \right)$
        $\bar{R} = \bar{R} + \eta\alpha (n_b)^{-1} \sum_{j=1}^{n_b} y_j$
    end
    if *τth step since last target network update* **then**
        $\mathbf{w}^- = \mathbf{w}$
    end
end

Algorithm 5: DQN with RVI Q-Learning

Input: The policy π to be used
Input: Differentiable loss function $loss\_fn$
Algorithm Parameters: Step sizes α
Algorithm Parameters: Reference sample size n_r
Algorithm Parameters: Batch size n_b
Algorithm Parameters: Number of steps between target network updates τ
Algorithm Parameters: Number of steps to train for T

Initialize action-value network Q with random weights $\mathbf{w}$
Initialize target network weights $\mathbf{w}^- = \mathbf{w}$
Initialize empty experience replay buffer $\mathcal{B}$
Obtain initial state S_1
for $t = 1$ **to** T **do**
    Select action A_t according to π
    Take action A_t, observe R_{t+1} and S_{t+1}
    Store experience $(S_t, A_t, R_{t+1}, S_{t+1})$ in $\mathcal{B}$
    if $t \geq n_r$ *and* $\mathcal{B}$ *has at least* n_b *experiences* **then**
        $Q_0 = (n_r)^{-1} \sum_{j=1}^{n_r} Q(S_j, A_j; \mathbf{w})$
        Let $(S_{i_j}, A_{i_j}, R_{i_{j+1}}, S_{i_{j+1}})_{j=1}^{n_b}$ be a random sample of n_b experiences from $\mathcal{B}$
        for $j = 1$ **to** n_b **do**
            $y_j = R_{i_{j+1}} + \max_a Q(S_{i_{j+1}}, a; \mathbf{w})$
            $\ell_j = loss\_fn(y_j, Q(S_{i_j}, A_{i_j}; \mathbf{w}^-))$
        end
        $\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} \left((n_b)^{-1} \sum_{j=1}^{n_b} \ell_j \right)$
    end
    if τ*th step since last target network update* **then**
        $\mathbf{w}^- = \mathbf{w}$
    end
end

Algorithm 6: DQN with R-Learning

Input: The policy π to be used
Input: Differentiable loss function $loss\_fn$
Algorithm Parameters: Step sizes $\alpha, \eta > 0$
Algorithm Parameters: Batch size n_b
Algorithm Parameters: Number of steps between target network updates τ
Algorithm Parameters: Number of steps to train for T

Initialize action-value network Q with random weights $\mathbf{w}$
Initialize target network weights $\mathbf{w}^- = \mathbf{w}$
Initialize empty experience replay buffer $\mathcal{B}$
Initialize average reward estimate $\bar{R} = 0$
Obtain initial state S_1
for $t = 1$ **to** T **do**
    Select action A_t according to π
    Take action A_t, observe R_{t+1} and S_{t+1}
    Store experience $(S_t, A_t, R_{t+1}, S_{t+1})$ in $\mathcal{B}$
    if $\mathcal{B}$ *has at least* n_b *experiences* **then**
        Let $(S_{i_j}, A_{i_j}, R_{i_{j+1}}, S_{i_{j+1}})_{j=1}^{n_b}$ be a random sample of n_b experience from $\mathcal{B}$
        for $j = 1$ **to** n_b **do**
            $y_j = R_{i_{j+1}} - \bar{R} + \max_a Q(S_{i_{j+1}}, a; \mathbf{w}^-)$
            $\ell_j = loss\_fn(y_j, Q(S_{i_j}, A_{i_j}; \mathbf{w})$
        end
        $\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} \left((n_b)^{-1} \sum_{j=1}^{n_b} \ell_j \right)$
        $greedy\_errors = \{y_j | A_{i_j}$ was the greedy action at timestep $i_j\}$
        $\bar{R} = \bar{R} + \eta\alpha \times avg(greedy\_errors)$
    end
    if τ*th step since last target network update* **then**
        $\mathbf{w}^- = \mathbf{w}$
    end
end