Comparing a Range of Reinforcement Learning Algorithms to Solve a Classic Rocket Trajectory Problem

Alireza Samar, Arnold Gomes, Danny Demarco, Joseph Dowling, Robert Williams, Tyler Christie
Department of Computer Science
University of Bath
Bath, BA2 7AY
{as4731, ag2342, dd480, jtd38, rdw41, tc905}@bath.ac.uk

1 Problem Definition

Application of reinforcement learning methods to modern artificial intelligence applications are both interesting and challenging, especially when environment transitions are unknown and there may be delayed rewards. This group project is an investigation into some of these methods and will attempt to train a number of agents to land a simulated spacecraft on the moon and compare their results for efficacy. To enable this, we use the LunarLander-v2 environment provided by OpenAI Gym (Brockman et al., 2016). The environment provides a classic rocket trajectory problem in which the rocket, starting at an elevated position is tasked with landing safely on a designated landing pad. There are two options in using this environment, discrete or continuous and we have chosen to use the discrete option. The 2d environment consists of the moon on the lower boundary with a section of it designated as the landing pad limited by the area between 2 flags, and an area of space above it in which the rocket may fly. The rocket itself is fitted with 2 legs to its base and three engines to provide thrust vertically or horizontally. To understand how the environment functions in more depth we can break it down into 4 main sections:

1. States
   (a) Starting state
   Each episode begins with the rocket in the same position; at the top, center of the viewport. The variable factor is that there is a random force applied to the rockets center of mass at the outset.
   (b) Observation space
   During an episode there are 8 possible states for the rocket to be in and consist of an 'x' and 'y' coordinate to denote its position, the linear velocity in both 'x' and 'y', the current angle of the rocket, the angular velocity, and two boolean values that represent whether each leg of the rocket is in contact with the ground or not. The landing pad is designated with (0,0) for its 'x' and 'y' coordinates.
2. Actions
   There are 4 discrete actions available at any point: no action, fire right engine (left orientation), fire left engine (right orientation), fire main engine (upward orientation). Fuel is considered infinite and there is no range of acceleration, engines are binary and either on or off.
3. Rewards
   (a) Positive Rewards
   The reward for succesfully moving from the starting state and coming to rest at the landing pad range between 100-140 points depending on the accuracy of the landing. Each leg that is in contact with the ground provides an extra 10 points to incentivise a

level approach to the landing pad. Once an episode is succesfully solved, a reward of 200 points is awarded.

(b) Negative Rewards
To incentivise the agent to land on the platform as quickly as possible, negative rewards are given for each use of any of the engines in each frame of an episode of -0.3. If the rocket crashes (touches the moon), it receives -100 points.

4. Episodes
Each episode starts in the given starting state and runs until the terminal state is activated. A terminal state is activated in any of the following instances:

   (a) The rocket moves outside of the viewport (moves too far left, right, or up)

   (b) The rocket crashes into the moon (misses the landing platform and hits the lower boundary)

   (c) A sleep state is reached, i.e. not in motion and not involved in a collision. This includes once the rocket has succesfully landed on the landing platform.

With these particulars in mind we can see that our agent is incentivised to direct the rocket from its starting position, to the landing pad with the minimal amount of engine use (fuel) as possible, while keeping the rocket stabilised as much as possible so that its legs may both make contact with the landing pad simultaneously. Incentivisation is produced by a large positive reward for reaching its platform, with bonus points for landing there in an upright position, and negative rewards for moving too far off course or colliding with the moons surface.

2 Background

Reinforcement learning is a powerful tool for decision making under uncertainty. Previous work has been done in solving the lunar lander environment using different techniques. This section will briefly introduce MDPs, Bellman Equation, Policy Gradient Methods, and other methods and then describe their strengths and weaknesses in solving the lunar lander problem.

Markov-decision processes (MDPs) are a type of mathematical model used in decision analysis and operations research. MDPs are used to model decision making environments (deterministic or stochastic) where information required for a decision is entirely contained in the previous state, not a history of states. In an MDP, the decision-maker is represented by an 'agent' who takes actions in a finite set of possible states, and a transition matrix determines the next state. The MDP framework allows an agent to learn a Markov policy; a state-action mapping function that maximises expected discounted future reward from a given state (Sutton, Precup and Singh, 1999).

MDPs find application in machine learning and control theory as a simplistic environment model. In machine learning, MDPs can be used to model the problem of learning how to take actions that maximize a reward. In control theory, MDPs can be used to model the problem of controlling a system to maximise or minimise a particular criterion, such as the expected value of the reward.

There are many methods for learning such a Markov policy, one of which is Q-learning. In this approach, a Q-table tabulates a set of state-action pairs, the values of which are updated via a Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma max_{a'} Q^*(s^{'}, a^{'})|s, a] \tag{1}$$

Dynamic programming, a class of solutions for solving such Bellman equations, finds its roots in optimal control theory. It is a technique for breaking down a problem into smaller subproblems, solving each subproblem, and then combining the solutions to the subproblems to solve the original problem.

There are a few different ways to implement dynamic programming for RL problems. For small state spaces, solutions may be tabulated. The table can be updated as the agent learns which actions lead to the best outcomes (Lewis and Vrabie, 2009). For intractable state-action spaces, a function approximator is used.

The current reward and discounted maximum future reward can be described as the optimal value for a given pair of action-state. Since the number of Q-values would be computed as finite, the typical approach to solving iteratively is to use dynamic programming:

$$Q_{i+1}(s,a) = \mathbb{E}[r + \gamma max_{a'} Q_i(s^{'}, a^{'})|s,a], \lim_{i\to\infty} Q_i = Q^* \tag{2}$$

In contrasted to value iteration approaches, policy gradient methods work by directly estimating a distribution that encodes a Markov policy.

Advantages of policy gradient methods include generally better exploration than ϵ-greedy methods due to the gaussian noise in their distribution (as well as optional addition of an entropy factor), and avoidance of problems like overestimation bias, found in Q-learning (Hasselt, 2010). Vanilla Policy Gradient (VPG) is an on-policy method (Peters and Schaal, 2006) amongst others we discuss such as TRPO, PPO and REINFORCE, and these methods are often less sample efficient than the off-policy, such as Deep Q-Network (DQN) (Mnih et al., 2015).

While the off-policy methods take advantage of experiences from previous policies, policy gradient methods generally reuse less experience. This difference describes why policy gradient requires significantly higher sample complexity to approach the same solution (Gadgil, Xin and Xu, 2020).

3 Method

It was decided that 3 algorithms would be implemented to train our agents so that the different approaches could be compared and contrasted for efficacy against this particular problem. The 3 algorithms selected were; REINFORCE, a traditional policy gradient method chosen as a baseline method, and the more modern DQN and PPO algorithms. To give a better understanding of our chosen method PPO, we will first introduce TRPO, as this is where this method is derived from.

3.1 TRPO

Trust Region Policy Optimization (TRPO) (Schulman et al., 2015) is a policy-gradient method for on-policy, online learning. It uses the concept of a 'Trust Region' in policy space, which permits the policy to move only within a specific 'trusted' bound per gradient ascent step. This mitigates the risk of an agent moving into a region of policy space from which it cannot recover from a sample collected under a poor policy.

An advantage estimate is calculated using a sampled discounted return under a policy π, parameterised by θ, from an episode rollout, and a noisy value function estimate from a function approximator. Importance sampling is used to calculate the policy gradient from the old policy with a different sample distribution, and as such achieves greater sample efficiency than vanilla policy gradient methods (which use current policy to compute gradient). The objective function and target for optimisation is:

$$\max_{\theta} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \tag{3}$$

$$\text{subject to } \hat{\mathbb{E}}_t \left[KL \left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] \leq \delta \tag{4}$$

Where KL is Kullback-Leibler Divergence (Joyce, 2011), and δ our trust region bound. Whilst the specifics of KL-Divergence are outside the scope of this project, it is essentially a statistical measure of the distance between two probability distributions; the policy update is subject to the constraint that KL-Divergence between the new and old policy must not exceed some hyperparameter δ for a gradient step. TRPO naturally falls into an Actor-Critic type model, where the value estimation is performed by the critic and the actor encodes the policy distribution.

3.2 PPO

Proximal Policy Optimization (Schulman et al., 2017) is a simplification and improvement over TRPO. Problems with TRPO were perceived to be complexity of implementation, and tractability

of computation with regards to the KL-Divergence bound, which is an expensive operation with second-order optimisation required. A simpler bound was found to be more effective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}^t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^t) \right] \tag{5}$$

Where $r_t(\theta)$ is identical to the probability ratio from TRPO ($\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$). The clip function replaces the KL-Divergence constraint, which clips the policy probability ratio such that the change does not exceed some hyperparameter ϵ, typically 0.2 (Schulman et al., 2017). In addition to this, PPO provides some non-trivial code-level optimisations which significantly improve performance over TRPO (Engstrom et al., 2020).

PPO was chosen for the proposed domain as it is relatively simple to implement, provides relatively good sample efficiency over other policy gradient methods, and is not particularly sensitive to hyperparameters (Schulman et al., 2017).

3.3 DQN

DQN (Deep-Q Networks) (Mnih et al., 2015) were the first foray into combining traditional reinforcement learning methods, Q-Learning (Watkins and Dayan, 1992), with deep neural networks. Revolutionary at the time, it allowed an agent to approximate the Q values of previously intractable large state/action spaces. Using these networks in place of the policy $\pi(s, a)$, or value $q(s, a)$ functions.

Observations from the environment are fed into the network, then training is performed via gradient descent with the aim of minimising a chosen loss function, typically Mean Squared Error. The algorithm functions similarly to Q-learning, however it also incorporates a replay buffer, target network for loss stability and function approximation for large state-action spaces. The target network is employed to provide a target for optimization as the algorithm suffers from instability caused by the presence of off-policy learning, function approximation and bootstrapping simulataneously. This is know as the deadly triad (Shangtong Zhang, 2021). DQN addresses this by using two separate networks; learning is performed with the prediction network, and then the target network is synchronised periodically. The replay buffer is used to store all transitions (S_t, A_t, R_t, S_{t+1}) encountered in the environment, then at every timestep a batch is sampled from these stored transitions and used to perform an update on prediction network parameters, defined as:

$$\begin{aligned} &\text{Set} y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \text{max}_{a'} Q(s_{t+1}, a'; \theta_2) & \text{for non terminal } s_{t+1} \end{cases} \\ &\text{Perform gradient descent on } \nabla_{\theta_1} L_\delta(y_j, \hat{q}_1(S_j, A_j; \boldsymbol{\theta}_1)) \end{aligned} \tag{6}$$

In which every C steps the target network θ_2 is updated to the local network θ_1.

DQN was chosen for its historical significance, as it has led to many further improvements in modern reinforcement learning, such as the direct descendant Hessel et al. (2018) an amalgamation of many individual improvements to DQN, which was state of the art for its time, or any other deep reinforcement learning method which have built of the work of Mnih et al. (2015). In addition to this it enjoys fast convergence and low sample complexity.

3.4 REINFORCE

The final method we chose for this environment was REINFORCE (Williams, 1992). REINFORCE is a traditional policy gradient method, but there are conflicting opinions about how exactly it should be implemented, possibly because as Williams (1992) defines it is fairly vague. The update function for network parameters θ is defined as (Sutton and Barto, 2018):

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\Delta_\theta \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \tag{7}$$

Where α is a hyperparameter controlling learning rate, G_t is our Monte-Carlo sampled discounted returns, and A_t, S_t are sampled actions and states respectively from a trajectory collected under policy π. By exploiting the likelihood ratio trick (Silver, 2015), we can rewrite this as:

$$\theta_{t+1} = \theta_t + \alpha G_t \Delta_\theta \text{log} \pi(A_t|S_t, \theta_t) \tag{8}$$

Notice that unlike PPO, this update method only takes into account current policy, and as such results in much lower sample efficiency. In many instances, due to the high variance of Monte-Carlo sampling methods, an arbitrary baseline $b(s)$ is subtracted from G_t in the update. $b(s)$ does not vary with a and as such reduces per-update variance.

The conflict surrounding implementation details for REINFORCE is with regard to the frequency of updates. Some accounts (Sutton and Barto, 2018) suggest updates to the network should be done at every timestep, Others (Silver, 2015) suggest updates on some time horizon T, and Williams (1992) suggests a number of update points, including per-episode (i.e. traditional Monte-Carlo) updates. We chose to implement REINFORCE with per-episode updates, and no baseline.

Using REINFORCE as a baseline policy gradient method, we are able to compare the results of our more modern PPO and DQN algorithms.

4 Results

Algorithms were coded in base PyTorch (Paszke et al., 2019). Due to time constraints, Bayesian Optimization was not carried out to discover optimal hyperparameters. As a result, hyperparameters were sourced from the original papers, or by consulting other online implementations and heuristic search. We make no claim that the hyperparameters used in the below experiments are optimal. Algorithms were run 10 times, and the graphs below are the average of these runs. Linear interpolation was used to fit data points per run to the average run length. Full graphs for every run can be found in Appendix A.

Full details of methodology and hyperparameter choices can be found in Appendices B and C respectively.

4.1 PPO

PPO solved the Lunar Lander environment in an average of 608 episodes with a standard deviation of 102.6:

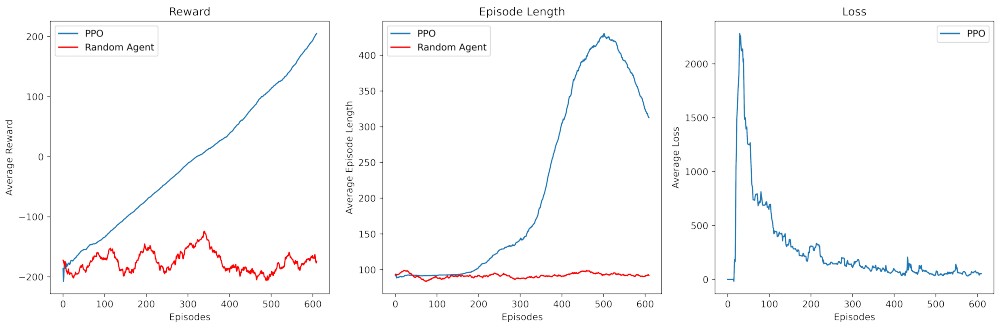

Figure 1: (left-to-right) Average reward, episode length and loss for PPO.

4.2 REINFORCE

REINFORCE only solved this environment adequately in 3 out of 10 runs. Non-terminating runs were cut off after 5000 episodes:

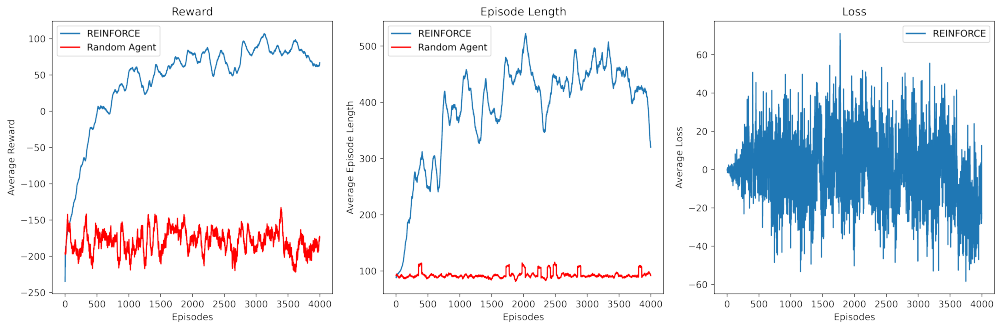

Figure 2: (left-to-right) Average reward, episode length and loss for REINFORCE.

4.3 DQN

DQN solved the Lunar Lander environment in an average of 311 episodes with a standard deviation of 77.6:

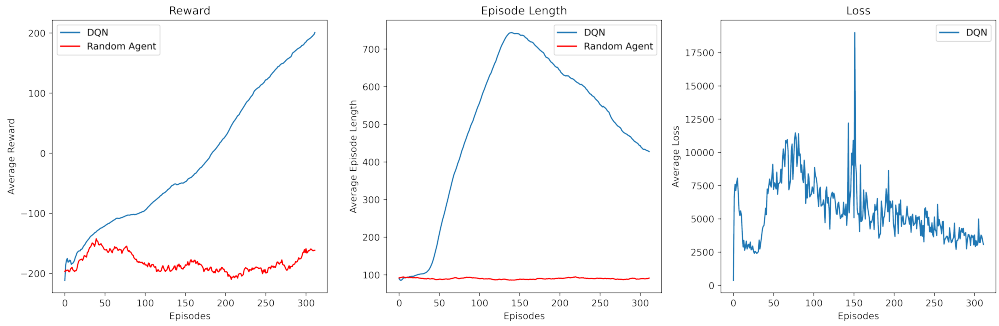

Figure 3: (left-to-right) Average reward, episode length and loss for DQN.

5 Discussion

Our results suggest that DQN is the best fit for this environment, with the shortest time to solve and the least deviation between runs. PPO is generally considered a more stable method due to the update function; the clip punishes actions that have a negative advantage that become more likely heavily, and rewards actions with positive advantage that become more likely only modestly. This is why we see a longer time to solve for PPO, as it is more cautious to back itself into a region of policy space. However, it appears we do not reap the benefits of PPO in such a simple environment. Interestingly, if we look at the individual runs for PPO and DQN as seen in Appendix A ,we see that PPO runs are grouped much more tightly throughout training and slightly diverge towards the end of training, approximately after reaching an average reward of 100. DQN at first glance appears to have more inter-run variance earlier on during training, and we even see one run with vastly different results to others. It is likely that, should we increase our sample size, we would see a much higher standard deviation between runs for DQN, as this is what we would expect. DQN tends to suffer from instability to a greater extent than PPO for a few reasons. Firstly, it is an offline method, and as such it learns from minibatches from the replay buffer. The replay buffer contains data from previous policies, and as such may introduce some irrelevant data into our loss calculations. This is a phenomenon known as distribution shift (Fu et al., 2019), and can make convergence behaviour hard to understand. Evidence for this is the outlier run seen in Appendix A, which follows a completely different trajectory to other runs despite having the exact same hyperparameters. Secondly, we have a non-stationary target for optimization; the target network, causing a ‘moving goalpost’ effect for optimisation. This can be mitigated with less frequent synchronisation between the target and prediction networks, however this is a purely empirical countermeasure and behaviour is not well understood in the literature.

Another feature of note is the distinction between episode lengths seen between Figures 1 and 3 for PPO and DQN respectively. We see that at its peak, PPO solution times are almost half the length of DQN. Moreover, if we again reference Appendix A, we see that episode lengths for PPO ramp up much more slowly than DQN. It appears that DQN is quickly finding what is arguably a suboptimal policy - slow landings with hovering, contrasted to what appears to be a more direct approach by PPO. The difference in speed can clearly be seen in the demonstration video supplied. This can be attributed to the Q-learning update, which simply takes the max over available actions. A max over noisy Q-value estimates supplied by the neural network is likely to result in overestimation bias, a property which has been demonstrated in Q-learning (Thrun and Schwartz, 1993; Hasselt, 2010), and is exacerbated by the use of function approximators. Overestimation of the 'hovering' behaviour seen with the DQN is likely causing the repetition of this behaviour and the lack of exploration in policy space.

The behaviour of REINFORCE in this scenario is what we expected, showing a trend of high variance. This variance is mainly due to the Monte-Carlo episodic nature of the updates. Unlike Temporal Difference methods (Tesauro et al., 1995), which use some portion of true sampled returns in addition to a bootstrap, Monte-Carlo methods use the true sample return as an update. As a result, updates will vary greatly as per-episode variance is high, and as such it is a fairly unstable learning method. REINFORCE is a true online method, which does not take into account the previous policy or any previous experience, and this is likely why we see REINFORCE converge on some runs and get stuck in local maxima on others.

6 Future Work

Given more time, more rigorous experimentation procedures would be followed, such as bayesian optimisation for hyperparameter tuning, and implementing pruning algorithms to determine optimal network architecture. Secondly, it would be interesting to apply some of the code-level changes seen in OpenAI's PPO implementation in Stable Baselines that are not mentioned in the paper. Baselines makes some non-trivial code-level optimisations such as value function clipping, reward scaling, orthogonal network initialisation and layer scaling which significantly improve performance, the likes of which are not discussed in the original paper (Engstrom et al., 2019), and may help us eke out some additional efficiency. As DQN was introduced many years ago now, there has since been considerable work spent analysing and improving the algorithm. For future work we would seek to implement the much improved and much more complicated state of the art improvement, Rainbow (Hessel et al., 2018), an algorithm built from six individual improvements appended and combined with DQN. Work has been performed analysing Rainbow and found that it is a large improvement over base DQN in our chosen environment. Then additionally, among the included improvements, distributional RL (Bellemare, Dabney and Munos, 2017) was the most necessary within this environment Ceron and Castro (2021). This is something we would like to examine and affirm in future work.

7 Personal Experience

Overall the team felt very positive about the project. Positive responses from all members indicate that completion of the works has deepened knowledge of reinforcement learning methods, while allowing us to work on a fun and challenging problem. The research completed to aid choosing appropriate algorithms gave us a deeper insight into how the field has developed over time and some of the benefits and drawbacks to different approaches. The team-working experience also ran very smoothly, with no conflicts and everybody participating eagerly to complete their allotted tasks and help each other out along the way. Specific difficulties faced during algorithm development were related mostly to the learning curve associated with PyTorch. The autograd engine is rather complex and takes some work to become proficient in its use. Understanding where to use commands such as, $torch.no\_grad()$, or how the $optimizer.step()$ and $loss.backward()$ operate, and are related, are specific pain points. Memory management is also hard to get-to-grips with, one team member reported receiving many memory leaks in which the cause could not be ascertained, and so had to rewrite code to clear the issue. The general view is that 20% of the effort was understanding the algorithm, and 80% was technicalities associated with PyTorch.

References

Bellemare, M.G., Dabney, W. and Munos, R., 2017. A distributional perspective on reinforcement learning. *International conference on machine learning*. PMLR, pp.449–458.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym. `arXiv:1606.01540`.

Ceron, J.S.O. and Castro, P.S., 2021. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. *International conference on machine learning*. PMLR, pp.1373–1383.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L. and Madry, A., 2019. Implementation matters in deep rl: A case study on ppo and trpo. *International conference on learning representations*.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L. and Madry, A., 2020. Implementation matters in deep rl: A case study on ppo and trpo [Online]. *International conference on learning representations*. Available from: `https://openreview.net/forum?id=r1etN1rtPB`.

Fu, J., Kumar, A., Soh, M. and Levine, S., 2019. Diagnosing bottlenecks in deep q-learning algorithms [Online]. In: K. Chaudhuri and R. Salakhutdinov, eds. *Proceedings of the 36th international conference on machine learning*. PMLR, *Proceedings of Machine Learning Research*, vol. 97, pp.2021–2030. Available from: `https://proceedings.mlr.press/v97/fu19a.html`.

Gadgil, S., Xin, Y. and Xu, C., 2020. Solving the lunar lander problem under uncertainty using reinforcement learning. *Southeastcon 2020*.

Hasselt, H., 2010. Double q-learning. *Advances in neural information processing systems*, 23.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2018. Rainbow: Combining improvements in deep reinforcement learning. *Thirty-second aaai conference on artificial intelligence*.

Joyce, J.M., 2011. *Kullback-leibler divergence* [Online], Berlin, Heidelberg: Springer Berlin Heidelberg, pp.720–722. Available from: `https://doi.org/10.1007/978-3-642-04898-2_327`.

Lewis, F.L. and Vrabie, D., 2009. Reinforcement learning and adaptive dynamic programming for feedback control. *Ieee circuits and systems magazine*, 9(3), pp.32–50.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. et al., 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540), pp.529–533.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S., 2019. Pytorch: An imperative style, high-performance deep learning library [Online]. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, eds. *Advances in neural information processing systems 32*. Curran Associates, Inc., pp.8024–8035. Available from: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Peters, J. and Schaal, S., 2006. Policy gradient methods for robotics. *2006 ieee/rsj international conference on intelligent robots and systems*. IEEE, pp.2219–2225.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P., 2015. Trust region policy optimization [Online]. In: F. Bach and D. Blei, eds. *Proceedings of the 32nd international conference on machine learning*. Lille, France: PMLR, *Proceedings of Machine Learning Research*, vol. 37, pp.1889–1897. Available from: `https://proceedings.mlr.press/v37/schulman15.html`.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. *arxiv preprint arxiv:1707.06347*.

Shangtong Zhang, Hengshuai Yao, S.W., 2021. Breaking the deadly triad with a target network. *International conference on machine learning*. PLMR, pp.12621–12631.

Silver, D., 2015. Lectures on reinforcement learning. URL: `https://www.davidsilver.uk/teaching/`.

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*. Cambridge, MA, USA: A Bradford Book.

Sutton, R.S., Precup, D. and Singh, S., 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2), pp.181–211.

Tesauro, G. et al., 1995. Temporal difference learning and td-gammon. *Communications of the acm*, 38(3), pp.58–68.

Thrun, S. and Schwartz, A., 1993. Issues in using function approximation for reinforcement learning. *Proceedings of the 1993 connectionist models summer school hillsdale, nj. lawrence erlbaum*. vol. 6.

Watkins, C.J. and Dayan, P., 1992. Q-learning. *Machine learning*, 8(3), pp.279–292.

Williams, R.J., 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3), pp.229–256.

Appendices

A All Runs For All Algorithms

PPO

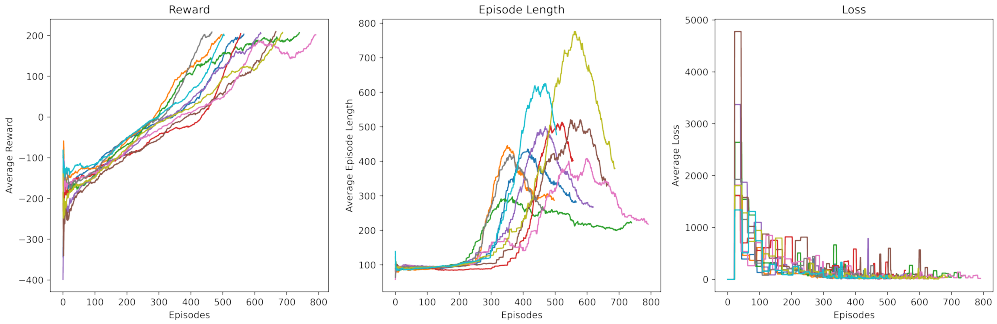

Figure 4: (left-to-right) Average reward, episode length and loss for PPO.

DQN

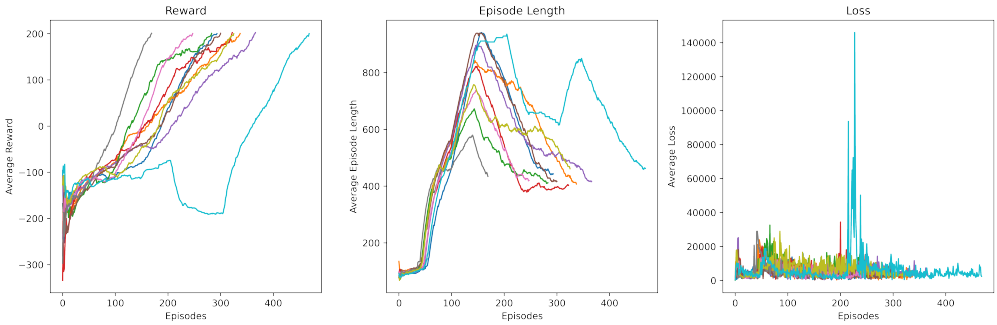

Figure 5: (left-to-right) Average reward, episode length and loss for DQN.

REINFORCE

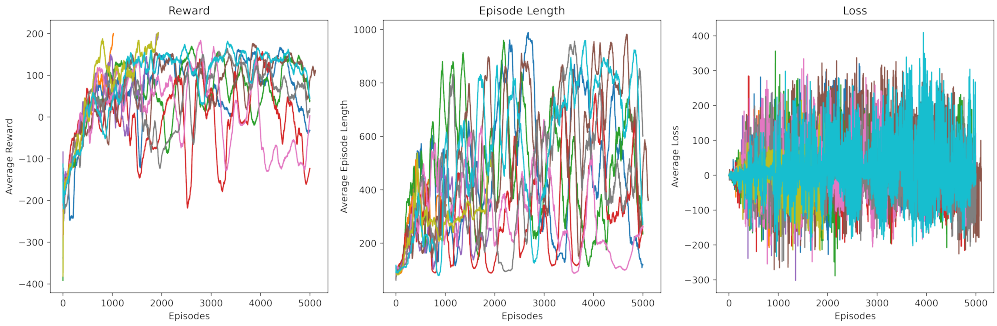

Figure 6: (left-to-right) Average reward, episode length and loss for REINFORCE.

B Methodology

Algorithms were run for a maximum of 5000 epochs, and were considered solved when the running 100 episode average reached an average reward of over 200, the point at which the environment is considered solved according to the specification. Each method was run 10 times and an average was taken. Logs were extracted from tensorboard.

C Hyperparameter and Architecture Choices

The network architecture used was chosen in accordance to recommendations in (Ceron and Castro, 2021). We used two fully connected layers with 256 neurons and relu activations, followed by the final output layer. We use the same network architecture for all methods to maintain consistency between experiments (PPO actor network and REINFORCE are the same structure, critic network for PPO is only 1 layer deep of the same size).

PPO

Hyperparameter	Value
discount factor γ	0.99
time horizon T	2048
batch size	2048
learning rate (actor)	3×10^{-4}
learning rate (critic)	1×10^{-3}
clip factor ϵ	0.2
gradient steps per batch	80

DQN

Hyperparameter	Value
discount factor γ	0.99
ϵ (start)	1.0
ϵ (end)	0.01
ϵ (decay)	1×10^{-4}
batch size	64
learning rate α	5×10^{-4}
target network sync	4 frames
replay size	100,000
warm start steps	64

REINFORCE

Hyperparameter	Value
discount factor γ	0.99
learning rate α	3×10^{-4}