ECE239AS (RL) Final Report: PPO Implementation for OpenAI Environments

Joshua Vendrow
Department of Computer Science
jvendrow@ucla.edu

Glen Meyerowitz
Department of Electrical Engineering
gmeyerowitz@ucla.edu

Rahul Malavalli
Department of Computer Science
rahul133@ucla.edu

1 Introduction

In this project, we are building upon the default project suggested for the class that examines the details and applications of the seminal Proximal Policy Optimization (PPO) Algorithms paper introduced by Shulman et al. [1]. The secondary aim of our project involves applying this PPO implementation to simple OpenAI environments, such as the CartPole-v1 and CarRacing-v0 environments.

1.1 Background

Reinforcement learning (RL) has used value-based methods, like Q-learning, to great success in certain applications. Deep Q-networks (DQNs) and their variations, for example, have exceeded human expert play in some Atari video games [2]. Value-based RL, however, can usually only be used to produce deterministic policies, and are not directly applicable to continuous action spaces. Value-based methods could adapt to continuous action spaces through some form of discretization, but they tend to perform poorly in these high-dimensional action spaces.

By learning the policy function directly, policy gradient algorithms attempt to remedy some of the concerns with value-based methods. Some of the immediate benefits of policy gradient methods include their inherent support for continuous action spaces and stochastic policies [3]. The stochastic policies were especially useful in probabilistic scenarios, like tic-tac-toe, where a deterministic policy is not preferred. Learning through stochastic policies also allows policy gradient methods to circumvent strategies like the epsilon-greedy policy, since a stochastic policy already explores the action space probabilistically.

However, the “vanilla" policy gradient suffers from its tendency to converge to a local optimum rather than the global one. Furthermore, because the policy is being learned directly, even small changes in the parameter space of the policy can result in large impacts to the performance of that policy. Therefore, vanilla policy gradient methods explore the policy space via very small steps to avoid collapsing the entire policy training process, making it much slower to train than normal Q-learning. These small step sizes greatly reduce sample efficiency; because the step size is small, policy gradient methods tend to utilize a relatively low amount of information from each sample it encounters. This type of sample inefficiency is especially undesirable when the algorithm has to gather samples by interacting with a real environment.

A new policy gradient algorithm called trust region policy optimization (TRPO) attempts to solve some of these problems. Specifically, TRPO attempts to improve sample efficiency by taking as large of a step as possible without affecting the policy too greatly. The TRPO algorithm does so by enforcing that each step taken must land within some distance, or “trust region", from the previous policy; here, “distance" is defined by the KL-divergence between the previous and new policies’

probability distributions [4]. This allows TRPO to take the maximum possible step size without leaving the trust region.

Although TRPO theoretically provides a good method, in practice the computation of KL-divergence per step proves to be too expensive and makes many applications intractable. Even after performing an approximation that makes the TRPO calculations efficient, it becomes difficult to select hyper-parameters (determining the coefficient for the KL-divergence penalty) that can generalize to many different problems, or even to different stages within one problem.

2 Proximal policy optimization

The PPO algorithm attempts to solve some of these problems by replacing the KL-divergence in trust regions with conservative policy iteration (CPI), which utilizes a probability ratio that denotes the change in a policy per step [5]. Here the probability ratio is defined as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{1}$$

where π_θ is the policy with parameter θ. In practice, this probability ratio can be calculated through repeated sampling of the environment. To overcome excessively large policy updates from the original CPI implementation, this is combined with a new "clip" functionality that attempts to prevent the algorithm from taking too large of steps.

To do so, PPO introduces a novel learning objective L^{CLIP}, presented below:

$$L^{CLIP} = \hat{\mathbb{E}}\Big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big] \tag{2}$$

where θ is the observed parameter, $\hat{\mathbb{E}}$ denotes an expectation calculated through discrete time steps, r_t is a probability ratio measuring the change in the new policy over the old policy, $\hat{A}_t$ is an estimator of the advantage function (where the advantage given an action and state is measured as the difference between the Q function for an action at a state and the value function of the state), and ϵ denotes the parameter by which to clip the probability ratio $r_t(\theta)$ to discourage too much change from the old policy. In Figure 6, we display a graph of this clip function for positive and negative advantages.

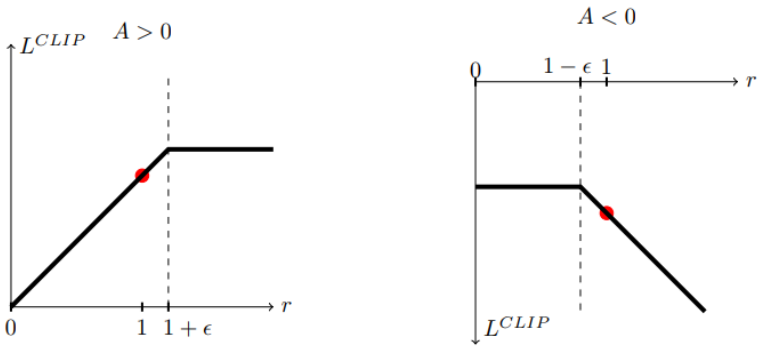


Figure 1: The plot of L^{CLIP} for positive advantages (left) and negative advantages (right) as a function of the probability ratio $r_t(\theta)$. We see that this loss function clips any benefits that cause the probability ratio to to exceed the range $(1-\epsilon, 1+\epsilon)$.

Combining this loss with a squared-loss error and an entropy term, PPO produces a $L^{CLIP+VF+S}$ loss function, produced below:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\Big[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\Big] \tag{3}$$

where θ is the observed parameter, L^{VF} is the squared-error loss $L^{VF} = \big(V_\theta(s_t) - V_t^{targ}\big)^2$ for the learned state-value function $V(s)$ from the advantage function calculation, S is an entropy function to ensure exploration as per [6][7], and c_1 and c_2 are constant coefficient hyperparameters.

Utilizing the composite objective function in equation (3), the PPO algorithm is able to generalize effectively to many different settings. It was able to outperform many continuous control environments, and even replicate state of the art results in some discrete domains.

In Algorithm 1 we display the basic pseudocode for the PPO algorithm. Here, L refers to the $L_t^{CLIP+VF+S}$ loss from above.

Algorithm 1 PPO

for iteration $= 0, 1, 2, ...$ **do**
  Run policy $\pi_{\theta_{old}}$ in environment for T steps
  Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  Optimize loss L wrt θ with K epochs of SGD (via Adam)
end for

This pseudocode presents a basic outline for the implementation. Over the course of some fixed or variable number of iterations, the actor is used to collect T timestamps of data, and then the model is optimized with respect to the $L_t^{CLIP+VF+S}$ for K epochs. In the next section, we fill out the details of our implementation choices on top of this outline.

3 Implementation

We implemented the PPO algorithm in Python with the PyTorch library. Our implementation accepts an OpenAI gym environment and optimizes a policy to run on this environment. We used an Actor-Critic model, where both the actor and critic are represented by neural networks.

3.1 Actor-Critic Methods

Policy gradient methods inherit many of the same limitations that value-based approaches, like Q-learning, suffer from due to the nature of reinforcement learning. For instance, both methods require some form of sampling to learn from an environment. The two most prominent sampling methods are Monte Carlo (MC), which generates samples only after an entire episode has completed, and Temporal Difference (TD) learning, which learns after each step in an episode by approximating the next step's value through its own value function. Thus, the MC method exhibits low bias but a very high variance, resulting in noisy gradients that can be especially harmful in policy gradient-based learning. Actor-critic models were introduced to take advantage of the lower variance and iterative nature of TD methods.

Specifically, actor-critic methods create two models that are trained simultaneously; the actor, which acts as the policy function, and the critic, which acts as the value function. This combination implements TD learning by using the critic (value function) to evaluate the value of an action taken by the actor (policy function). Utilization of actor-critic models with PPO helps mitigate some of the issues related to high variance that vanilla policy gradient methods are typically susceptible to [8].

3.2 Generalized Advantage Estimation (GAE)

Policy gradients will often have a high variance, making it difficult to reduce loss and increase reward consistently. By implementing an Advantage Function, it is possible to dramatically decrease the variance of the policy gradient, but they introduce some amount of bias into the algorithm [9]. Schulman et al. provide the definition of Generalized Advantage Estimation (GAE), which estimates the advantage using the λ-return, as:

$$\hat{\mathbb{A}}_t = R_t(\lambda) - V(s_t) \tag{4}$$

The GAE value is calculated and updated during each step of execution. Finally, the GAE is fed back to the agent loss calculation as a part of agent training during each step.

4 Rudimentary Environment

To gain a better understanding of environment creation, we initially experimented with a rudimentary car racing track that we created ourselves.

4.1 Initial Environment

Specifically, each simplified race track is generated on a 2D plane via a function of the form $y = f_i(x)$, where y is the y-coordinate of the track center at some x-coordinate x. When $f_i(x)$ is applied to all $x \in [x_{min}, x_{max}]$, a center line for the track is generated, where x_{min} and x_{max} are the bounds for the 2D plane. A different $f_i(x)$ is defined for each unique i^{th} track. The environment is also determined by a "track width" parameter, which determines the perpendicular distance of the track sides from the generated center line. The figure 2 shows three separate generated tracks for a) $y = 0$, b) $y = x$, and c) $y = sin(x)$. All environments have a track width of 2 units.

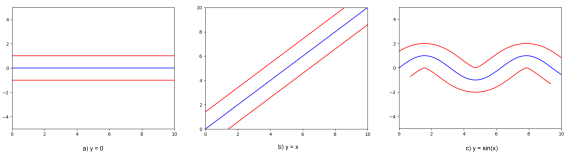


Figure 2: Graphs of rudimentary race track examples.

4.2 Initial State and Action Space

The self-driving car is modeled by an agent with a discrete state and action space. The state space combines the "sensed" features from the race track environment with the car's steering angle (with respect to the 2D plane) and the speed. The car's action space, however, is limited to only manipulations of the steering angle and speed, also discrete. The car's internal state is a velocity defined as a tuple (v, θ), where v is the speed and θ is the angle. Each action is also a tuple $(\Delta v, \Delta\theta)$, where Δv and $\Delta\theta$ are applied to the state's speed and angle if allowed by the environment; for example, the speed is capped at a maximum absolute value of 1. The allowed values for each of these variables are included in table 1 below:

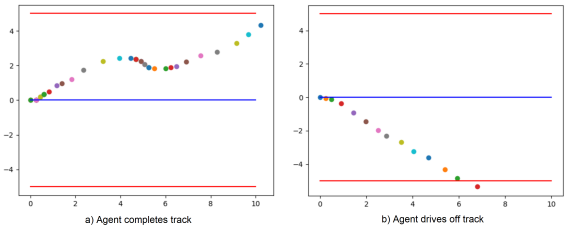
Table 1: Allowed Values

v	$0.00, \pm 0.25, \pm 0.50, \pm 0.75, \pm 1.00$
θ	$0, \pm 15, \pm 30, \pm 45$
Δv	$-0.25, 0, +0.25$
$\Delta\theta$	$-15, 0, +15$

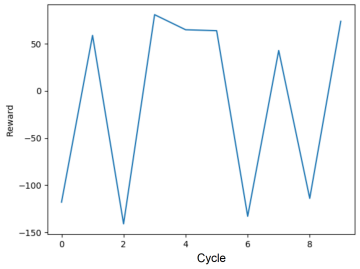
4.3 Initial Baseline

We create a baseline to compare our trained algorithm, and to ensure that we are properly learning in the environment. To do so, we implement an agent that randomly chooses an action from the action space in each time instance. For evaluation, we also have generated a simple reward function that yields $+100$ if the track is completed, -100 if the agent drives off the track, and -1 for each time step that has passed without either other event.

We first define the state of the agent at $t = 0$. The agent is located at $(0, 0)$, with a speed of 0 and a direction parallel to the x-axis. The following plots show the motion of the agent as it moves along the track. In a) the agent successfully completes the track, while in b) the agent drives off the track.



The following plot shows the reward plot for a single epoch with the random agent.



5 Experiments

5.1 CartPole-v1

To test our PPO implementation, we apply it to the CartPole-v1 OpenAI environment. As illustrated in figure 3, the CartPole-v1 environment includes a “cart" agent that attempts to balance a long pole by driving forward or backward (right or left) on a flat surface; the goal of the agent is to ensure that the pole does not fall off of the cart.

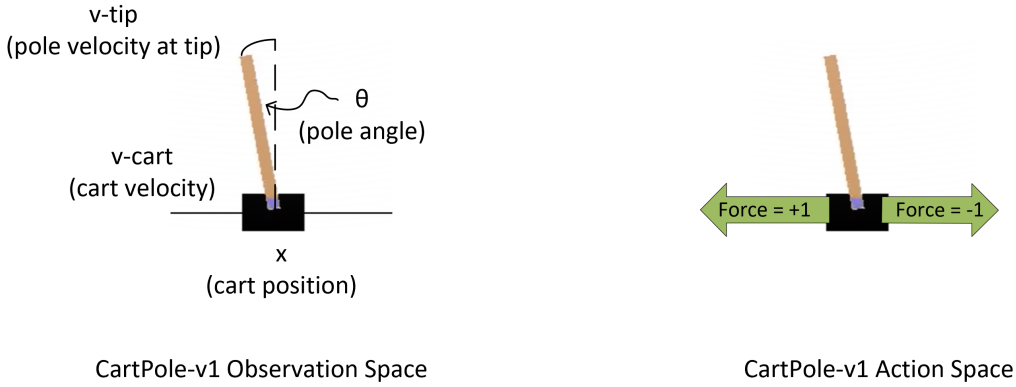


Figure 3: Observation space (left) and action space (right) for the CartPole-V1 environment.

The environment returns an observation space of four continuous values (cart position, cart velocity, pole velocity at tip, and pole angle), and supports two discrete actions (drive right or drive left). In line with the goal of balancing the pole as long as possible, the environment returns a reward of +1 for every timestep that it is upright, and terminates an episode when the pole is more than 15 degrees off of the vertical or the cart itself is more than 2.4 units from the center. The results of this experiment are described in section 6.1.

5.2 Discretized CarRacing-v0

To test our PPO on a self-driving car, we opted for the CarRacing-v0 environment from OpenAI, because of its well-developed implementation. Specifically, the CarRacing-v0 environment exposed a 96x96 RGB image of a top-down view of the car and race track as its observation space, and supported a continuous action space containing three values (steering angle, gas, and brake). An example of this observation space is visible in figure 4, and the range of accepted action space values is described in table 2.

Table 2: CarRacing-v0 Original Continuous Action Space

Action Value	**Minimum**	**Maximum**
Steering Angle	-1.0	+1.0
Gas	0.0	1.0
Brake	0.0	1.0

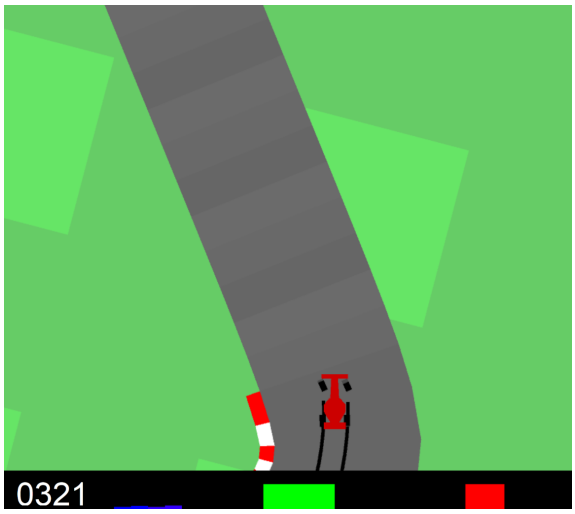


Figure 4: Observation space for the CarRacing-v0 environment.

To simplify the training process, we first discretized the action space such that each action value can only correspond to the ones included in table 3, with three possibilities for each value. As a result, we created a total of 27 discrete actions from all combinations of possible action values. We chose a discrete action space because a continuous action space would require the PPO algorithm to sample an action to perform from a probabilistic policy for every step in the training, which may require much more time to successfully complete due to an emphasis on exploration.

Table 3: Discretized Action Space

Steering Angle	**Gas**	**Brake**
-1.0	0.0	0.0
0.0	0.5	0.5
+1.0	1.0	1.0

6 Results

6.1 CartPole Results

We developed two metrics to quantitatively evaluate the performance of and measure the effectiveness of our PPO policy. First, we calculate average loss by averaging the loss over the optimization epochs for the sample collected at each iteration of the PPO algorithm. We calculate average reward by

pre-selecting a series of initial observations and averaging the total reward of the policy with these initial observations. In Figure 6 we display the average loss and reward at each iteration of our experiment.

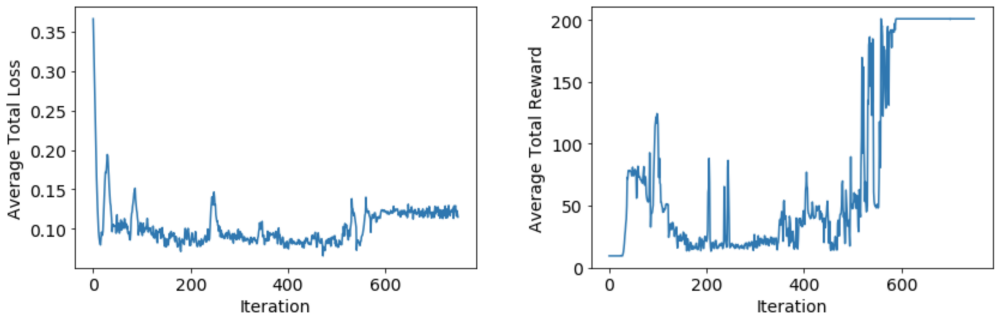


Figure 5: Average loss (left) and average reward (right) for the CartPole-V1 environment. When calculating average rewards, we terminate each trial after 200 steps because the agent will have successfully "solved" the environmental criteria after this number of steps. We calculate these rewards separately from the optimization task using a pre-selected series of initial observations, so this sampling termination does not affect our policy.

We see that after about 600 iterations of the algorithm, we are able to consistently achieve a reward of 200, satisfying the solve criteria. We also note that the increase in loss after iteration 500 can be attributed to updates in the critic, which evaluates state values to calculate advantages.

We implemented separate Actor-Critic models for each of the environments on which we trained an agent using the PPO implementation. The following figures show the neural network architectures that were used for both the actor and critic for the CartPole-v1 environment. While the network architectures used were the same, the size of the input and output layers are different due to the different environment sizes.

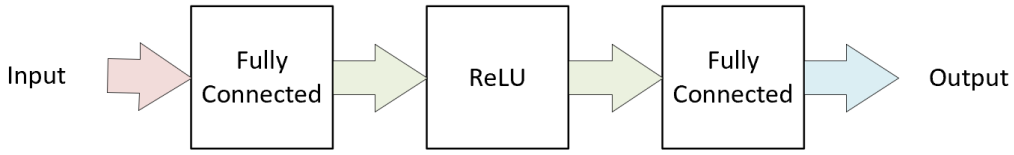


Figure 6: Neural Network Architecture for the Actor and Critic Models in the discrete CartPole-v1 Environment. The input for both the actor and critic neural network is the current state of the agent. For the last fully connected layer, the critic has one output and the actor's output size is the number of actions. For the actor model, we perform classification by adding a softmax layer before the final output.

6.2 CarRacing Results

When implementing a model for this modified CarRacing-v0 environment, the image-based observation space could no longer be effectively solved via the fully-connected neural networks used for the CartPole-v1 environment. Instead, convolutional neural networks (CNNs) had to be leveraged because of their success in image analysis tasks. Therefore, we explored two main approaches: training a new CNN from scratch, and fine-tuning a pretrained ResNet-18 CNN with an appended fully-connected neural network.

Unfortunately, even when trained for over 1500 episodes, both policy (actor) models ended up settling on only one action for all states, regardless of its efficacy in the actual environment. We attribute some

of this to vanishing gradients we observed during training, that may be mitigated with the addition of batchnorm layers, experimentation with different learning rates, and/or other hyperparameter fine-tuning.

7 Conclusion

Our goal for this project was to build on the default project suggested for the class that examines the details and applications of the seminal Proximal Policy Optimization (PPO) Algorithms paper introduced by Shulman et al. [1]. In order to engage with this, we decided to develop a simple environment in which we could train an agent to interact with the environment using a PPO algorithm. Our end goal was to train an agent to drive on a racetrack utilizing our basic PPO implementation.

We were able to successfully implement a PPO algorithm based on the paper from Shulman et al., which used actor-critic models to optimize behavior and a Generalized Advantage Estimation in order to reduce variance in the model.

We were able to successfully apply the PPO algorithm to environments with discrete action spaces, such as the CartPole-v1 environment from OpenAI, which utilized a fully-connected neural network that converged in less than 1000 PPO steps. In CartPole-v1, over the limited number of training iterations, we were able to see an increase in the reward received by the agent from 50 to 200.

To apply the discrete PPO implementation to an OpenAI car racing environment, we modified the CarRacing-v0 by discretizing the action space. We trained two separate actor-critic models in this environment, one that used a new CNN and a second that used a pretrained ResNet-18 model. With this implementation, both actor-critic models suffered from vanishing gradient issues and lack of computational resources. Further model experimentation, hyper-parameter tuning, and prolonged training (via GPUs) may be necessary to obtain better results.

We learned more about the CarRacing-v0 environment and it presented additional challenges beyond what we had thought. Traditionally, the CarRacing-v0 environment generates a 2D box of pixels that must be interpreted by a CNN. To promote model learning, we may be able to simplify the environment by explicitly extracting features, such as steering angle, speed, and distance to the nearest wall. This could allow us to utilize a simpler fully-connected neural network that could be trained faster.

Computational resources have been a major limiting factor in these results. Although we were able to locally train simple fully-connected neural networks for environments like CartPole-v1, we quickly realized that the training larger CNN-based models would require hardware acceleration (GPUs, etc.) to be effective.

We learned a tremendous amount about RL and policy optimizations methods while working on this project. We extend our thanks to Dr. Yang, Joris, Qiujing, and Chinmay for their help this quarter.

8 Team Contributions

Rahul Malavalli discretized the CarRacing environment from OpenAI to allow for the PPO implementation to interface with that environment. He generalized the actor-critic method and PPO implementation to work on environments of all sizes and continuous vs. discrete action spaces; such as for the CarRacing-v0 environment. He set up and experimented with the pretrained ResNet-18 CNN approach to the CarRacing-v0 environment. He contributed to our Poster Presentation and Final Report.

Glen Meyerowitz developed a custom environment for car racing outside of the OpenAI framework. The team was not able to test the PPO algorithm on this environment, as we did not get beyond the OpenAI environments. He contributed to our Poster Presentation and Final Report.

Joshua Vendrow implemented the Proximal Policy Optimization (PPO) for our agent, including the actor-critic method and GAE. He spent many hours troubleshooting the implementation on both the CartPole and CarRacing environments from OpenAI to ensure proper behavior of the algorithm. He contributed to our Poster Presentation and Final Report.

Code

Here is a link to the GitHub repository containing our implementation:

`https://github.com/rahulm/ece239-deepracer`

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[3] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

[5] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, page 267–274, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[6] 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.

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

[8] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *ICLR*, 2017.

[9] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *ICLR*, 2016.

[10] Christopher J.C.H Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[11] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. *British Library*, 1989.