Benchmarking Agents Across Diverse Tasks

Alexander Moore
Data Science
Worcester Polytechnic Institute
Worcester, United States
ammoore@wpi.edu

Brian Lewandowski
Computer Science
Worcester Polytechnic Institute
Worcester, United States
balewandowski@wpi.edu

Jannik Haas
Data Science
Worcester Polytechnic Institute
Worcester, United States
jbhaas@wpi.edu

Quincy Hershey
Data Science
Worcester Polytechnic Institute
Worcester, United States
qbhershey@wpi.edu

Scott Tang
Data Science
Worcester Polytechnic Institute
Worcester, United States
stang3@wpi.edu

***Abstract*—Understanding which models succeed and why at which tasks is a foundational experience in reinforcement learning. Following a baseline of techniques and best practices found in the literature this project will show a comparison of multiple reinforcement learning techniques applied in a few common environments. In particular, this project implements several algorithms under the Q-learning, policy learning, and actor-critic categories and compares the results across several tasks. We demonstrate the strengths and weaknesses of a large sample of models on discrete-action and continuous action tasks. This work explores a set of baseline results for diverse reinforcement learning algorithms, in order to compare and contrast the performance of models on diverse tasks.**

***Index Terms*—Q-Learning, Policy Learning, Policy Gradient, Actor-Critic, Reinforcement Learning**

I. Introduction

Project 3 explored how to construct a Deep Q Network (DQN) or a similar model by following the classic DQN algorithm outlined originally in [3]. Building off of this recent experience this project takes the deep learning techniques discussed in class and applies them to several well-known tasks using Q-learning, policy learning, and actor-critic methods. Our group implemented, optimized, and benchmarked a large variety of models from these diverse families in order to analyze and discuss the success and failures of each model. We quantify model success by comparing the scores achieved as well as the time-to-convergence in terms of training episodes needed to reach a threshold score. The results of these methods are compared and discussed.

The openai gym [1] was chosen to be used as the game environment. Two games were chosen to be used for comparison of the algorithms in this paper. First, the Breakout game was used for its familiarity from Project 3 and that it has a known level of difficulty. In addition, it serves as a discrete action space environment. The Breakout-v0 game is a productive demonstration as part of the poster presentation since the students in the class are all familiar with the process of implementing, optimizing, and testing a DQN-family model on the game. This will contextualize both the work of the project as well as expected score results, for example 50 for a baseline and 400 for the best-scorers in the course.[2] To contrast with this, the MountainCar was also chosen as it has both a discrete and continuous action space environment available for use. Using these environments multiple agents were implemented, trained, and tested using DQN, Double DQN, Dueling DQN, Multistep DQN, Multistep Double DQN, REINFORCE, Proximal Policy Optimization (PPO), and Advantage Actor Critic (A2C). A breakdown of where these algorithms are categorized in the field of reinforcement learning can be seen in Figure 1.

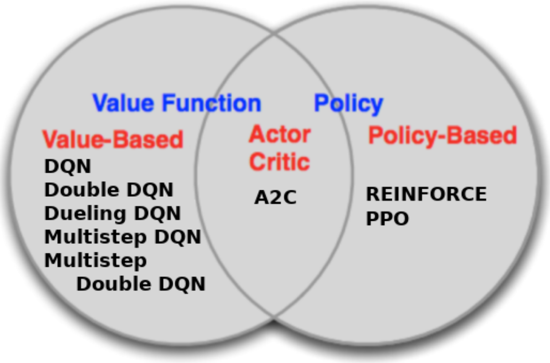

Fig. 1. **Implemented Agents in the field of Reinforcement Learning** A breakdown of where the implemented agents in this paper are categorized in the wider field of reinforcement learning.

This work shows that these algorithms all have strengths and weaknesses in a wide variety of areas including task completion, complexity, training time, and sensitivity to hyperparameters. Ultimately, the success of models is left largely to the hyperparameter selection and tuning steps.

The remainder of this paper is organized as follows:

- Section II provides background information regarding the methods implemented.

[1] https://gym.openai.com/

[2] https://github.com/yingxue-zhang/DS595CS525-RL-Projects/tree/master/Project3

- Section III provides a description of the methodology used to train, execute, and assess the algorithms explored.
- Section IV provides a discussion of the training process and associated results.
- Finally, the paper concludes in section V.

II. Background Information

A. Deep Q Learning (DQN)

Deep Q learning uses a deep neural network to learn coefficients ω such that the network's value function evaluates $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ state(S), action(A), reward(R) pairs stored in a memory replay buffer, improving the model's task reward. For our Breakout environment this will be a convolution over a stack screen space samples, and for Mountaincar the model is a simple fully-connected network over the game state: the car position and car velocity.

$$(R_t + \gamma_t \underset{a'}{max} Q_{\bar{\Theta}}(S_t, a') - Q_{\Theta}(S_t, A_t))^2$$

Deep Q Learning is trained by minimizing the loss taken by evaluating the Q(s,a) value between the agent's state-value function and a target Q function, which is not optimized but occasionally updated to keep the training Q learning.

B. Double Deep Q Learning (DDQN)

Double deep Q networks address overestimation bias and instability in the traditional deep Q learning model by dividing the action selection and evaluation processes into two networks. Value estimation is carried out by a more stable network that is periodically updated from the primary action selection network. This process inhibits bias and will generally converge faster or outright outperform DQN on all tasks.

$$Y_t{}^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{argmax} Q(S_{t+1}, a; \Theta_t), \Theta_t{}^{-})$$

C. Double Deep Q Learning ex Action Separation

As a variant to traditional DDQN networks, allowing action selection to be conducted by the value estimation network for target Q estimation adds additional stability to the network which may provide greater network stability and robustness towards higher learning rates versus the traditional algorithm.

$$Y_t{}^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{argmax} Q(S_{t+1}, a; \Theta_t{}^{-}), \Theta_t{}^{-})$$

D. Dueling Deep Q Learning

Dueling DQN separates the Q-learning process into two functions, the sum of the state and state-action estimator models. This approach ideally learns how to relatively value states by accounting for a learned state-value function $V(s)$, as well as an advantage function $A(s, a)$ interpreted as the value of taking action a while in state s. For this reason this different approach will be interesting to analyze on tasks where the actions might not always directly affect the environment, for example in breakout where many movements have little direct affect on the game environment.

Listing 1
Basic Policy Gradient The basic policy gradient algorithm

```
Loop until stopping criteria met:
    for a batch of episodes:
        - Step through episode
        - Store reward gained

    - Determine reward gradient of batch
    - Update policy parameters
```

E. Multistep Deep Q Network Models

As opposed to single step Deep Q Learning, multi-step accrues rewards over n-step periods with rewards accumulated over that period with gamma discounting before being applied in a similar DQN structure.

$$R_t{}^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t{}^{(k)} R_{t+k+1}$$

$$(R_t{}^{(n)} + \gamma_t{}^{(n)} \underset{a'}{max} Q_{\bar{\Theta}}(S_{t+n}, a') - Q_{\Theta}(S_t, A_t))^2$$

F. Basic Policy Gradient

Policy gradient algorithms in general are methods that learn a parameterized policy that can select actions without consulting a value function [7]. A value function may be utilized or learned to support learning a given policy, however, it is not involved in the process of choosing an action with these methods [7]. The Basic Policy Gradient algorithm treats learning of the policy as an episodic case using a performance measure that equates to the reward obtained following the current policy given a starting state of an episode. At its core, the Basic Policy Gradient algorithm takes the gradient of expected rewards for an episode and uses it to update the policy parameters. This can be summarized by the two equations below,

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} R(\tau^n) \nabla log \pi_\theta(a_t^n | s_t^n)$$

$$\theta = \theta + \eta \nabla \bar{R}_\theta$$

where N is the number of episodes, T is the steps within an episode, R is the reward for a given episode, θ are the policy parameters, a is a given action, and s is a given state.

Implemented in pseudocode, this algorithm would be similar to Listing 1. While not implemented as part of this project is is the building block for all subsequently developed policy gradient models.

G. REINFORCE Policy Gradient

The REINFORCE Policy Gradient method also referred to as Monte-Carlo Policy Gradient makes a slight adjustment to the Basic Policy Gradient algorithm of using the return from each time step instead of the overall reward from the episode

Listing 2
REINFORCE Policy Gradient The reinforce policy gradient algorithm

```
For each episode from the current policy:
    For t=1 to T−1 do:
        {$\theta = \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) G_t$}

    end for
end for
return {$\theta$}
```

in the update of the policy parameters. This will reduce the high variance of the standard basic policy gradient algorithm. This change can be seen in the pseudocode in Listing 2.

H. PPO Policy Gradient

Unlike the Basic and REINFORCE policy gradient algorithms, the Proximal Policy Optimization (PPO) algorithm is an off-policy method, where the performance assessment and improvement of a policy is different than the one used for action selection (or sampling). The separate policies allow the collected data to be used more than once, which result in faster performance. Furthermore, typical policy gradient algorithms perform updates only once per sample while PPO can perform multiple updates with minibatches of each sample. PPO also addresses other shortcomings of the Basic Policy Gradient by subtracting the state-value as the baseline to improve update accuracy when rewards are non-negative via an Advantage function. This function also serves to reduce large variance without increasing bias. Finally, PPO improves stability by imposing a constraint on the distance (or difference) between the old and new policies by clipping their ratio to a small interval around one. Due to this constraint, some critics believe that PPO should be considered an on-policy method.

There are different options for implementing the PPO algorithm, as outlined in [6]. The one chosen for this work is known as the Actor-Critic Style but with only one actor. The psuedocode is depicted in Listing 3. The Actor-Critic method, specifically A2C, is described in the next subsection. In addition, entropy bonus and clipping (without the KL divergence penalty) of the surrogate objective function and a decaying learning rate were implemented in this work. Finally, the estimation of advantages followed the Generalized Advantage Estimation (GAE) approach described in [5].

I. Advantage Actor Critic

Actor-Critic methods, similar to the REINFORCE algorithm, learn both a policy function and state-value function. The main difference between them, however, is the fact that actor-critic methods use the value function as a way to bootstrap the value estimation. The implementation for this work follows closely to the ”One-step Actor-Critic” algorithm outlined in [7] with some practical updates for implementation and stands as a synchronous adaptation of the Asynchronous Advantage Actor critic algorithm (A3C) presented in [2]. It is

Listing 3
PPO Policy Gradient, Actor-Critic Style

```
for iteration = 1, 2,... do:
    for actor = 1, 2,..., N do: (N = 1 in this implementation)
        Collect dataset based on policy $\pi_{\theta old}$ for $T$ steps
        Compute advantages ($\hat{A}_1, ..., \hat{A}_T$) using
            Generalized Advantage Estimation (GAE)
    end for
    Optimize surrogate objective $L$ wrt $\theta$ by taking $K$ steps
        with minibatch (size $M \leq T$) SGD (Adam)
    $\theta_{old} \longleftarrow \theta$
end for
```

a temporal difference algorithm that relies solely on the current step for learning. This is in contrast to the many of the other algorithms implemented in this work as they involve utilizing a replay buffer for batch learning.

In addition, it should be noted that this implementation remains a basic implementation and does not include the commonly seen updates added to this method such as an additional entropy loss.

III. Methodology

A. Tools and Framework

This project used the Python language and several python libraries. In addition to the standard libraries, the following additional libraries were used:

- OpenAI Gym [1]
- PyTorch
- NumPy
- Matplotlib
- Pandas

The framework provided from Project 3 was built off of and adapted to support additional environments aside from Breakout. A framework was built such that the main training and testing pipeline was consistent across algorithm implementations. This allowed for each team member to focus on implementing specific models and algorithm learning details.

The general workflow using this framework can be seen in Listing 4.

It should be noted that all agents inherit a set of functions from a base class allowing for a small barrier to entry in setting up new agents. An agent runner class was constructed such that it handles the main training loop used by all algorithms implemented while the learning details are held within the specific agents themselves.

In addition, during model training, this framework stores all arguments used to start the session as well as periodic interim models and statistics in both raw format and as plots. This allowed for repeatability and easy metrics collection during every session. An example view of what this archive directory looks like can be seen in Listing 5.

Listing 4
ALGORITHM DEVELOPMENT WORKFLOW THE BASIC WORKFLOW USED TO DEVELOP NEW MODELS AND ALGORITHMS.

```
# Create a new model in models directory
# - Implement the __init__ function
# - Implement the forward function

# Create a new agent in the agents directory
# - Implement the make_action function
# - Implement the can_train function
# - Implement the train function

# Execute training for model on command line
# python main.py \
#     --train_dqn \
#     --model sample_model.SampleModel \
#     --agent sample_agent.SampleAgent \
#     --other_arguments
```

Listing 5
MODEL TRAINING ARCHIVE A TYPICAL ARCHIVE FROM A MODEL THAT HAS BEEN TRAINED FOR 2000 EPISODES.

```
archive/
   |
   test_learning_rate/
       |
       args.pkl
       1000_model.pth
       1000_optimizer.pth
       1000_training_metrics.csv
       1000_training_reward_plot.png
       1000_test_metrics.csv
       1000_test_reward_plot.png
       2000_model.pth
       2000_optimizer.pth
       2000_training_metrics.csv
       2000_training_reward_plot.png
       2000_test_metrics.csv
       2000_test_reward_plot.png
```

B. Gym Environments

Several different environments from the openai gym python packages were utilized on this project. A description of each of these environments and a description of the task to be solved by agents within them follows.

1) BreakoutNoFrameskip-v4: The Breakout environment involves interacting with the Atari 2600 game by the agent receiving image data for what has been seen on screen. After pre-processing handled by the atari wrapper code the state space consists of 4 stacked 84x84 images. An example of the original environment that would be viewed by a human playing the game can be seen in Figure 2.

The action space for this environment consists of 4 possible discrete actions:

- LEFT - Move agent left
- RIGHT - Move agent right
- NOOP - Do nothing
- FIRE - Places ball in motion at start of new lives

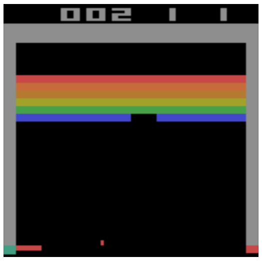

Fig. 2. **Example of the Breakout environment.** The Atari Breakout environment provided by openai.

This environment was chosen as it was familiar from Project 3 and sufficiently difficult for naive models. In addition, it serves as a well-contained discrete action space for all algorithms under test.

The reward structure for this environment consists of the traditional values for the blocks in the game where the first two levels provide one point, the next two levels provide 3 points, and the final two levels provide 7 points. For the learning agent, the atari wrapper environment clips this reward to be just one point for a timestep where a brick was broken.

2) MountainCarContinuous-v0: The Mountain Car environment in both its original and continuous form consists of the agent being a car beginning in the valley between two mountains. In this classic control task, the agent must apply a specific force to the mountain car with the goal of reaching the top of the right hand mountain as seen by the flag in Figure 3.

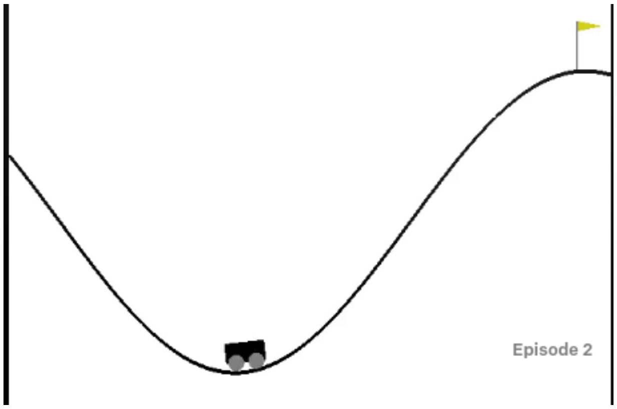

Fig. 3. **Example of the Mountain Car Environment** The mountain car environment provided by openai.

In this environment, the agent must learn to gather enough momentum by first going backwards a bit prior to going forwards in order to have enough acceleration to reach the flag.

The state space for this environment consists only of the mountain car's position and velocity. The action space for this environment is a single continuous action which is the force to be applied to the mountain car. A positive value is a force towards the goal and a negative value for this force puts the mountain car in reverse.

The reward structure for this environment consists of the agent receiving a small negative reward for the number of actions taken and a positive reward of 100 when reaching the goal. This environment is considered solved once a score of 90 is achieved [3].

This environment was chosen as it is a well-suited introductory environment to explore continuous action spaces. The game is significantly smaller than the breakout game in terms of state size, as well as being significantly easier to solve, as the optimal strategy is significantly less complex than the space of Breakout strategies. We expect the training process of this game to be significantly faster than Breakout, letting us train a greater number of agents and optimize the hyperparameters of these agent significantly more than in the Breakout setting.

3) MountainCar-v0: The MountainCar-v0 environment is the discrete action-space relative of MountainCarContinuous-v0 as described above. This environment entails the same general principles and overall goal but the action space has been altered to use discrete actions as listed below.

- PUSH LEFT
- PUSH RIGHT
- NO PUSH

The reward structure for this environment consists of -1 for each time step until the goal is reached. The environment is considered solved if an average of -110 points is achieved over 100 consecutive trials [4].

C. Continuous Action Spaces

One of the key goals for this project was to interact in an environment with continuous action spaces. In general, models that operate in discrete action spaces take as input the state space (or state-space and action) and return either a value function or policy function. The value function represents the value of the particular state and the policy function generally provides an action or probabilities for which actions to execute.

When dealing with continuous action spaces, this is shifted such that the policy learned is the mean and standard deviation for desirable actions within a continuous action space. As the agent learns, the standard deviation generally starts out wide and then narrows once it is more confident in the actions to take. Choosing an action consists of sampling from the learned distribution.

D. Continuous Advantage Actor Critic Implementation

The actor critic implementation follows closely to the ”One-step Actor-Critic” algorithm outlined in [7] with some practical updates made for implementation adapted from Phil Tabor’s public codebase [5]. This implementation is a temporal difference algorithm using no replay buffer and using only the current time step information to facilitate learning. In addition, it is implemented in an on-policy manner.

Listing 6
Continuous Actor Critic Model The model architecture used for the continuous actor critic implementation.

```
Fully Connected:   2 Input,512 Output
ReLU

|
v

Fully Connected:  512 Input, 512 Output
ReLU

|\
| \
|  \
|   |
|   v
|  Value Output Layer:  512 Input; 1 Output
|
v

Policy Mean
Output Layer:   512 Input;  1 Output
tanh

Policy Standard Deviation
Output Layer:  512 Input;  1 Output
Softplus
```

The target for this model was the MountainCarContinuous-v0 environment so the model and agent were structured with this in mind. One can see the model structure and hyperparameters of the final model in Listing 6 and Table I respectively.

TABLE I
Continuous Advantage Actor Critic Hyperparameters The hyperparameters used for training the continuous actor critic agent.

Hyperparameter	Value
Batch Size	Does not apply
Learning Rate	0.00015
Replay Buffer Size	Does not apply
Minimum Buffer Training Size	Does not apply
Starting Epsilon	Does not apply
Final Epsilon	Does not apply
Epsilon Decay Steps	Does not apply
Gamma	0.99
Loss Function	-log_loss * advantage
Optimizer	Adam
Target Network Update Interval	Does not apply

E. Continuous REINFORCE Policy Gradient Implementation

The REINFORCE Monte Carlo Policy Gradient algorithm was implemented to closely follow the pseudocode in [7]. To maintain consistent comparisons between the Policy Gradient algorithms, the same model as the Actor Critic implementation for the MountainCarContinuous-v0 environment was used

[3] https://github.com/openai/gym/wiki/MountainCarContinuous-v0

[4] https://github.com/openai/gym/wiki/MountainCar-v0

[5] https://github.com/philtabor

which can be found in Listing 6. The REINFORCE model simply discarded the Value Output Layer and produced only the mean and standard deviations for the action as output. The hyper parameters for this implementation can be found in Table II

TABLE II
CONTINUOUS REINFORCE HYPERPARAMETERS THE HYPERPARAMETERS USED FOR TRAINING THE CONTINUOUS REINFORCE AGENT.

Hyperparameter	**Value**
Batch Size	Does not apply
Learning Rate	0.000015
Replay Buffer Size	Does not apply
Minimum Buffer Training Size	Does not apply
Starting Epsilon	Does not apply
Final Epsilon	Does not apply
Epsilon Decay Steps	Does not apply
Gamma	0.99
Loss Function	-log_loss * return
Optimizer	Adam
Target Network Update Interval	Does not apply

F. Continuous PPO Implementation

As previously stated, our PPO implementation follows the Actor-Critic style. As a result, the structure of the model is identical to that of the one depicted in Listing 6. Otherwise, the remaining parts of the algorithm are almost identical to the one outlined in the [6]. Some modifications include the hyperparameter settings and the omission of the KL divergence penalty and a decaying clipping parameter. The hyperparameters for our implementation of the continuous version of the PPO algorithm are shown in Table III.

TABLE III
CONTINUOUS PPO HYPERPARAMETERS THE HYPERPARAMETERS USED FOR TRAINING THE CONTINUOUS PPO AGENT.

Hyperparameter	**Value**
Steps (T)	128
# of Actors	1
Batch Size (M)	128
Minibatch Size	4
# of Minibatches	32
Network Update Epochs (K)	4
Learning Rate	0.00015
Learning Rate Decay	Does not apply
Replay Buffer Size	Does not apply
Discount Factor (γ)	0.99
GAE Parameter (λ)	0.95
Prob. Ratio Clipping (ϵ)	0.2
Clipping Decay	Does not apply
Value Function Loss Coef.	0.5
Entropy Coef.	0.01
Loss Function	Surrogate Objective
Optimizer	Adam

G. Discrete Advantage Actor Critic Implementation

The discrete version of the actor critic algorithm adapted the continuous version discussed earlier for the BreakoutNoFrameskip-v4 environment. This included incorporating a known model baseline previously shown effective in this environment while adapting it for returning both the policy and values necessary for the actor critic method.

Listing 7
DISCRETE ACTOR CRITIC MODEL THE MODEL ARCHITECTURE USED FOR THE DISCRETE ACTOR CRITIC IMPLEMENTATION.

```
CNN:  32 8x8 filters , stride 4
ReLU

|
v

CNN:  64 4x4 filters , stride 2
ReLU

|
v

CNN:  64 3x3 filters , stride 1
ReLU

|
v

Fully Connected: 3136 Input; 512 Output
ReLU

|\
| \
|  \
|   |
|   v
|  Value Output Layer:  512 Input; 1 Output
|
v

Policy Output Layer: 512 Input;  4 Output
SoftMax
```

The model structure and hyperparameters for this implementation can be seen in Listing 7 and Table IV respectively.

TABLE IV
DISCRETE ADVANTAGE ACTOR CRITIC HYPERPARAMETERS THE HYPERPARAMETERS USED FOR TRAINING THE DISCRETE ACTOR CRITIC AGENT.

Hyperparameter	**Value**
Batch Size	Does not apply
Learning Rate	0.0003
Replay Buffer Size	Does not apply
Minimum Buffer Training Size	Does not apply
Starting Epsilon	Does not apply
Final Epsilon	Does not apply
Epsilon Decay Steps	Does not apply
Gamma	0.99
Loss Function	-log_loss * advantage
Optimizer	Adam
Target Network Update Interval	Does not apply

H. Discrete REINFORCE Policy Gradient Implementation

The REINFORCE algorithm was also implemented for discrete action spaces, specifically for the openai BreakoutNoFrameskip-v4 environment with the model outlined in Listing 7, and again discarding the value output layer.

I. Discrete PPO Implementation

The discrete version of the PPO algorithm heavily leveraged the continuous version discussed earlier. The major difference is the incorporation of a convolutional neural network (CNN) based model identical to the one used by the discrete Actor Critic algorithm as depicted in Listing 7. The hyperparameters for our implementation of the discrete version of the PPO algorithm are shown in Table V.

TABLE V
DISCRETE PPO HYPERPARAMETERS THE HYPERPARAMETERS USED FOR TRAINING THE DISCRETE PPO AGENT.

Hyperparameter	**Value**
Steps (T)	2048
# of Actors	1
Batch Size (M)	2048
Minibatch Size	4
# of Minibatches	512
Network Update Epochs (K)	4
Learning Rate	0.0003
Learning Rate Decay	-2.75e-8 / step
Replay Buffer Size	Does not apply
Discount Factor (γ)	0.99
GAE Parameter (λ)	0.95
Prob. Ratio Clipping (ϵ)	0.2
Clipping Decay	Does not apply
Value Function Loss Coef.	0.5
Entropy Coef.	0.01
Loss Function	Surrogate Objective
Optimizer	Adam

IV. RESULTS

A. DQN-Family Breakout Results

With the set of 5 DQN-family models (DQN, Double DQN, Dueling DQN, Multistep DQN, and Multistep DDQN), we did an additional investigation into the importance of the learning rate. For this reason we both benchmark the 5 models with the same low learning rate, then do a comparative experiment of a the same set of models where the only change is a significantly higher learning rate.

In training the DDQN ex Action Separation demonstrated much greater robustness to learning rate differentials, while tolerating much larger learning rates than the other models for some speed advantage against similarly structured DQN based models. Among models with total separation in action and value estimation, only the DDQN with multi-step (n=3) model provided enough model stability to train at similarly high rates (1.5e-4), albeit with weaker results over the first 10,000 episodes. At lower learning rates (0.0001) nearly all of the models were able to demonstrate progress towards convergence, training efficiently to various degrees over the first 10,000 episodes. The best performers in Breakout at the lower learning rate were the DDQN, DDQN ex Action Separation and Dueling DDQN.

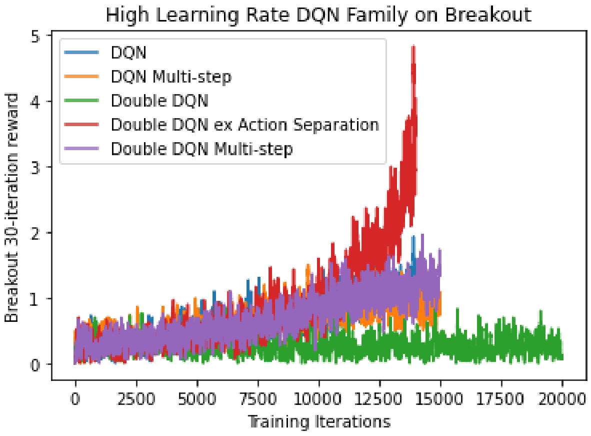

Fig. 4. The five models show large differences in the ability to quickly improve when using a large learning rate. We see that Double DQN without Action Separation significantly departs from the pack in terms of performance. Though Double DQN fails to learn in 20,000 iterations, the model is implemented correctly, shown by the performance in the low-learning-rate trials.

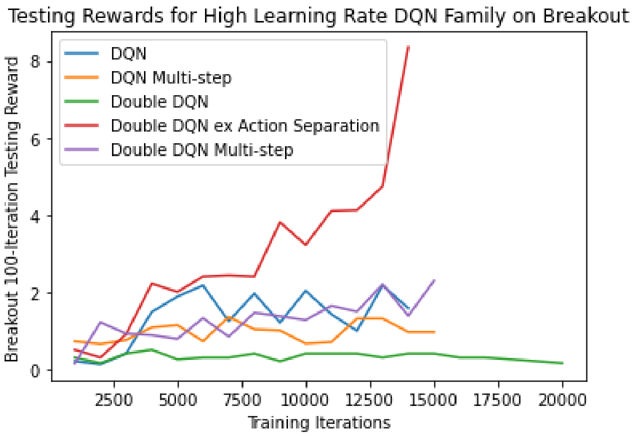

Fig. 5. The testing performances shown here and consistent with the training rewards: Double DQN without action seperation drastically outperforms all others in terms of rapid learning acceleration in Breakout.

B. DQN-Family MountainCar Results

For the MountainCar implementation of the DQN family, we expected a larger difference than what we observe here. It is possible that the simple nature of the game in terms of both the state and optimal policy makes it hard for some models to succeed drastically more than others. To a suprising degree, we see the training rewards of all three DQN models highly consistent, both in terms of the initial acceleration in learning as well as the moment and height of the plateau around 1000 iterations to a score around -120.

C. Continuous Advantage Actor Critic

The continuous actor critic agent was exercised using the MountainCarContinuous-v0 environment. This agent as im-

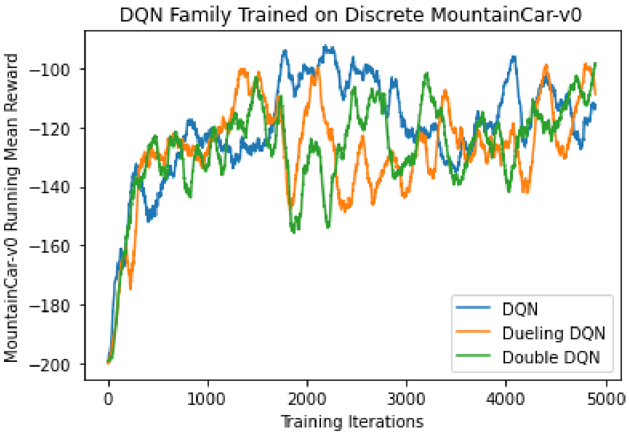

Fig. 6. The three DQN models implemented on the discrete MountainCar game show remarkably similar behavior in training.

plemented was able to obtain adequate scores in this game as well as meet the benchmark of 90 that is considered beating the game. This can be seen in the test reward plot in Figure 7. In this plot scores near 80 are achieved for starting at episode 7000 with the highest score coming in at 91.06.

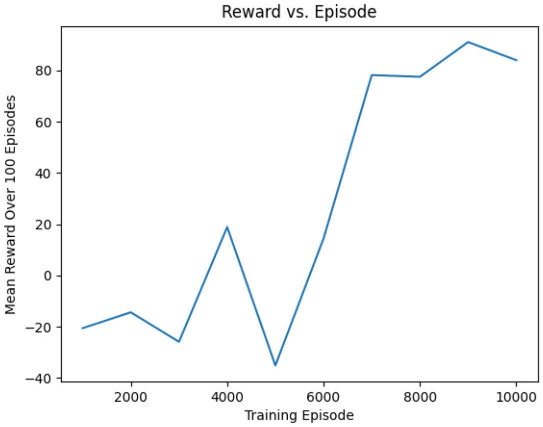

Fig. 7. **The actor critic agent playing MountainCarContinuous-v0** The actor critic agent successfully learns how to win the MountainCarContinuous-v0 environment.

Figure 8 displays the general steps learned by the agent to gain enough momentum to reach the goal.

This environment was run multiple times with this same agent to confirm consistency of results. While this type of result was achieved consistently there were also runs that displayed some of the known issues with the stability of the actor critic method. Looking at the training plot from a run in Figure 9, it can be seen that the agent begins to learn, then ultimately fails to learn after 5000 episodes of training.

This can be compared to the plot of a typical successful training of the agent as seen in Figure 10. In this plot one can see a pronounced general upward trend in the learning which has essentially converged around 9000 episodes.

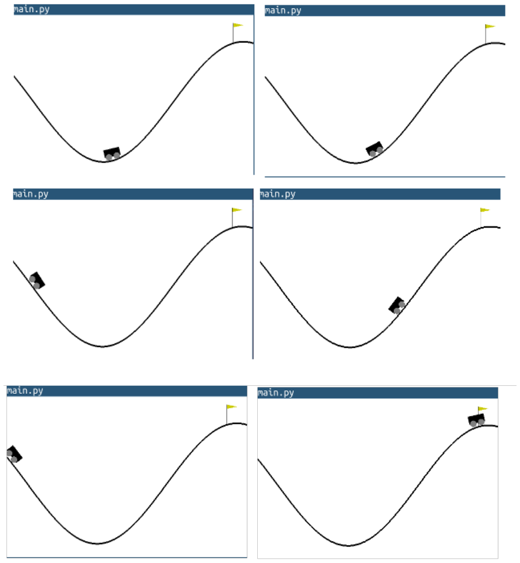

Fig. 8. **Typical Strategy learned to win Mountain Car.** The actor critic learned this strategy to win mountain car by rocking back and forth to gain enough momentum to reach the goal.

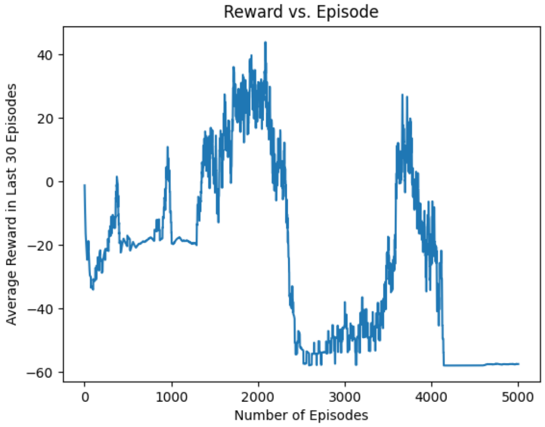

Fig. 9. **A bad training session for actor critic agent playing MountainCarContinuous-v0** The actor critic method has stability issues that can affect training.

D. *Continuous REINFORCE Policy Gradient*

The REINFORCE algorithm was unable to achieve great results in the MountainCarContinuous-v0 environment. As seen in the training reward plot Figure 11, the agent is unable to achieve the score of 90 required to ”solve” the game. In this case it learned to simply stay still to avoid the penalty of moving. The Monte Carlo methods suffer from a high variability and combined with the scarce reward in this environment, it was unable to overcome these obstacles. If the agent never stumbles upon the reward at the top of the hill, it will only know the penalty for moving and will therefore try to minimize the penalty instead of getting to the reward. A low learning rate was implemented to encourage more exploration, however it was unable to achieve the goal. Some modifications

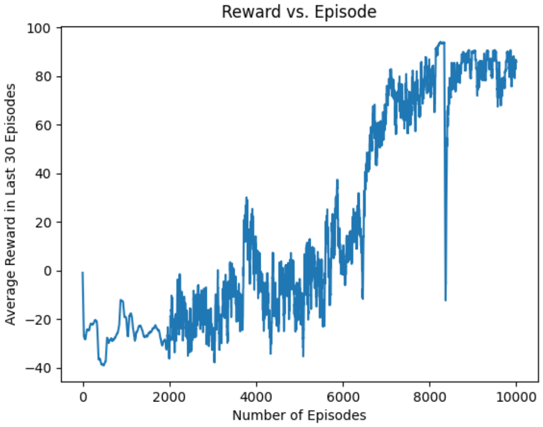

Fig. 10. **A typical training session for actor critic agent playing MountainCarContinuous-v0** A typical training session for the actor critic method.

to the reward system were attempted such as adding to the reward based on the y-coordinates or height of the agent on the hill, but the final results unfortunately remained the same.

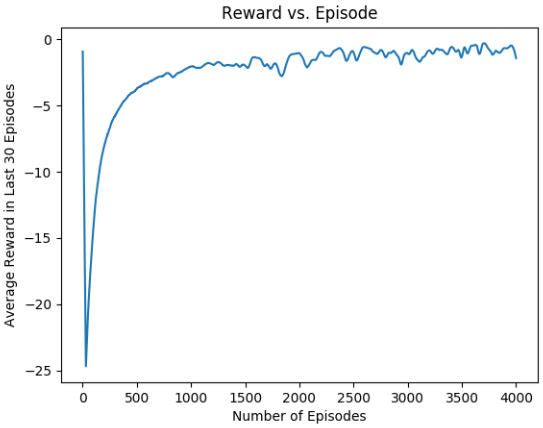

Fig. 11. **The training plot for a REINFORCE agent playing MountainCarContinuous-v0** A training plot for the REINFORCE algorithm showing it was unable to learn the optimal policy.

E. Continuous PPO

The continuous PPO agent as implemented could not obtain acceptable scores when trained and exercised in the MountainCarContinuous-v0 environment. The agent repeatedly and consistently plateaued to scores of slightly below zero. In other words, the continuous PPO agent learned that the highest scores were achieved if it moved as little as possible and remained in the bottom of the mountain because every step is a negative reward. It never realized that there is a much larger reward available at the top. This occurrence can be seen in the typical training reward plot in Figure 12. An examination of the loss and entropy during and between the model updates did not reveal any unusual behaviors. The loss was low with the occasional spikes and entropy decreases with time, as expected. Loss and entropy for a typical training session are depicted in Figure 13.

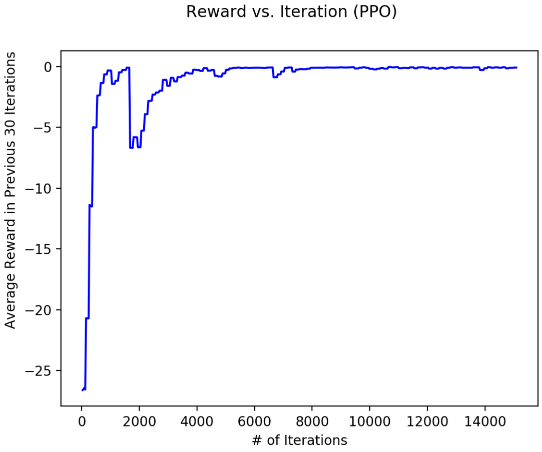

Fig. 12. **Typical Training Reward Plot for the PPO Agent.** A typical training session for the PPO agent playing MountainCarContinuous-v0.

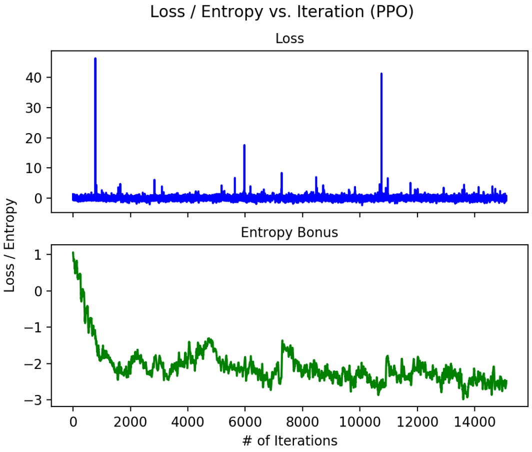

Fig. 13. **Typical Training Loss and Entropy Bonus Plots for the PPO Agent.** The loss and entropy during a typical training session for the PPO agent playing MountainCarContinuous-v0.

The plateau in the performance of the continuous PPO agent was also consistent when several hyperparameter values were modified, including the size and quantity of minibatches, and learning rates below a certain threshold. Figure 14 depicts the case where a learning rate of 0.003 is clearly too high since the mean rewards were consistently very low. All learning rates below 0.0003 did not improve the score, nor did they have any detrimental effects.

F. Discrete Advantage Actor Critic

The discrete advantage actor critic agent was trained and executed on the BreakoutNoFrameskip-v4 environment. Through this training we were unable to see significant learning taking place. The actor critic methods can be sensitive to hyperparameters such as learning rate so a hyperparameter search through

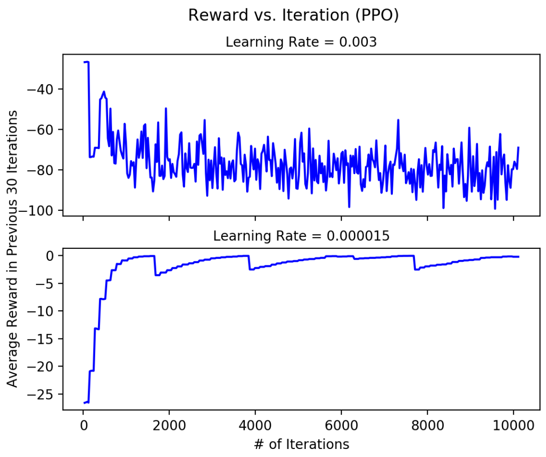

Fig. 14. **Learning Rate Comparison for the PPO Agent.** A learning rate of 0.003 was too high and performance suffered as a result when compared to a lower rate of 0.00015.

multiple learning rates was performed though none of them seemed to affect the learning of the agent. A summary of these failed attempts can be seen in Figure 15.

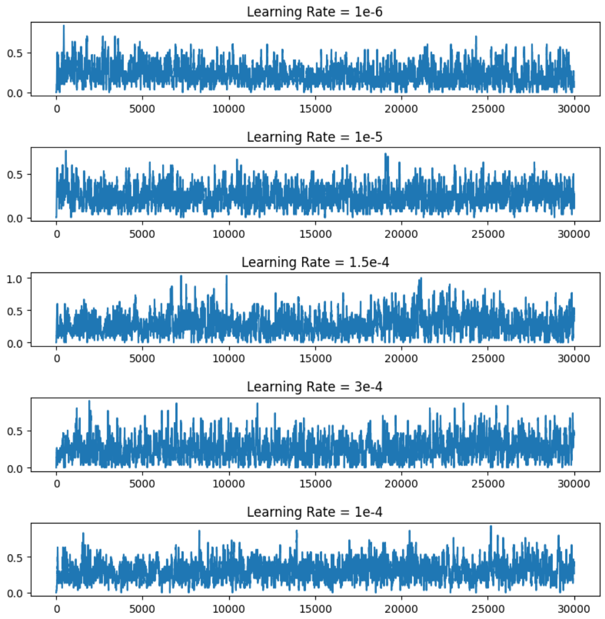

Fig. 15. **A search through various learning rates for actor critic on Breakout.** Changing the hyperparameter for learning rate did not improve the results for the discrete version of actor critic on the Breakout environment.

While better results were expected overall for this agent, its implementation lacked essentially all of the hallmark improvements made in [3] and [4] such as usage of a replay buffer. It is suspected that in a future work one may be able to make some of these adjustments such as use of a replay buffer and see improved results.

This result was interesting as it showed an algorithm that performs well in a continuous action space environment did not translate well to one with discrete actions. One factor affecting this, however, may be the increased complexity of the game of Breakout compared to that of the mountain car problem.

G. *Discrete REINFORCE Policy Gradient*

For the discrete version of REINFORCE we chose the popular BreakoutNoFrameskip-v4 environment. The agent was unable to learn a good policy for this environment and we attribute this to the simplicity of the REINFORCE algorithm as well as the complexity of the Breakout environment. We tested several different hyperparameters, but they followed the same trend as the Actor Critic plots in Figure 15. This implementation also did not include some optimizations such as adding a baseline, which we hope to implement in the future to see better results.

H. *Discrete PPO*

The discrete PPO agent as implemented was unable to learn and achieve an acceptable score when trained and exercised in the BreakoutNoFrameskip-v4 environment. The average scores achieved by the agent consistently bounced between 0 and 0.7, with an average of approximately 0.3. Its performance was flat and displayed no improvements even after training more than 20,000 iterations, as can be seen in the typical training reward plot in Figure 16. An examination of loss and entropy during and between the model updates did reveal a couple of unusual behaviors, as depicted in Figure 17. In general, the loss was low and very slowly decreasing, but a very noticeable pattern that correlates with the length of each horizon (or batch size) emerged. This artifact will need to be investigated further in the future. Entropy also slowly decreases with time, as expected, but there is a noticeable ”hump” around iteration 10,000. The entropy bonus is used by the agent to balance exploration with exploitation. The shape of this entropy plot might indicate that it is still mostly in the exploration stage.

A decaying learning rate that began at a reasonably low value of 0.0003 and continued at a rate of 2.75e-8 per iteration did not improve the performance of the PPO agent. Due to resource constraints, a factorial experiment with the hyperparameters was not possible. It is highly likely that improved hyperparameters could result in higher scores. Additionally, there could be features not considered in this implementation that might improve its performance in this environment, including a decaying clipping parameter, the use of multiple actors and the inclusion of a replay buffer.

V. Conclusion

Our discussed findings focus on two primary conclusions: the meta-learning about the process of a research project such as, as well as the findings of results on models.

First, we did get interesting results on the comparison of models on our two tasks. Especially when comparing to

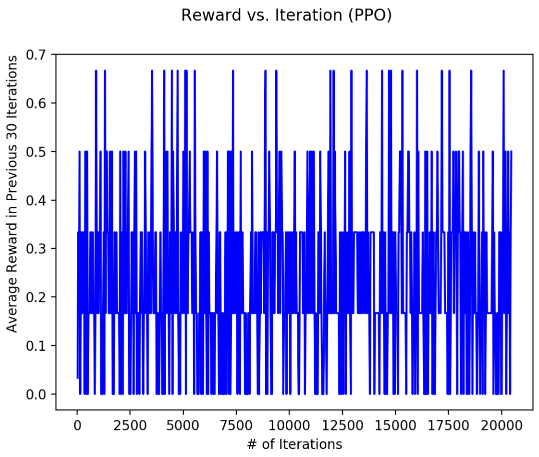

Fig. 16. **Typical Training Reward Plot for the PPO Agent.** A typical training session for the PPO agent playing BreakoutNoFrameskip-v4.

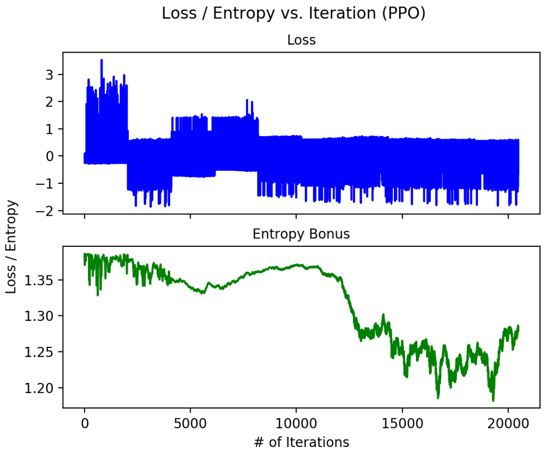

Fig. 17. **Typical Training Loss and Entropy Bonus Plots for the PPO Agent.** The loss and entropy during a typical training session for the PPO agent playing BreakoutNoFrameskip-v4.

Project 3, we find that without exhaustive training and parameter tuning, none of our breakout models were competitive with the kinds of scores found to be possible in the leaderboards. This should contextualize the primary importance of tuning over model selection, and the concession that more complex models do not necessarily outperform simpler models, especially when implementation time, tuning time, and training time are all considered. The training figures demonstrate this complication: though implemented, discrete advantage actor-critic, PPO, and others all failed to get off the ground due to the complications of implementation.

Additionally, the environments selected are as meaningful to the hypothesis as they are hard to select. It is hard to select two or more environments as experimental subjects that can well-discern the differences between models. For example in machine learning, two models can be directly compared over some selection of parameter searching, and quantify their variance and accuracy. However in reinforcement learning, training times for optimization over some set of parameters might be intractable, as well as quantification of their performance can be additionally challenging: are we interested in time-to-convergence? Variability in rewards? Mean reward in testing? These challenges are only exacerbated by the selection of additional games, where the innate complexity of reinforcement learning and the combination of policy, value functions, and agent training all drastically affect the outcome of the model's understanding of the game. For this reason, a few amended research directions are proposed.

As discussed, the scope of models proposed for this project was very wide. A selection of five distinct discrete-action agents as well as four policy gradient agents were all individually implemented from scratch. The additional complication of attempted development over two tasks only complicated this process further. Our goal as discussed in section I, was to compare the performance of models across tasks, with an additional goal of bench-marking models on each task. The outcome of this ideal proposal would be twofold: a set of results showing with some confidence the relative quality of models on particular results, as well as the pre-trained models which earned those results for future usage. Secondly, this project would show how agents change in *relative* quality as the task changes. Ideally, this comparison could be made over a large set of very diverse tasks: from discrete and continuous actions, large and complex state spaces to very simple ones (such as MountainCar), and strategies of simple and complex nature. This could be a fascinating result to have even an approximation of, which would serve both novice reinforcement learning students with a brief understanding of what works and why, as well as for experts seeking a benchmark for agents which generally thrive at certain tasks, such as learning simple strategies very quickly.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Volodymyr Mnih, Adri Puigdomnech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[5] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.