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. Results sentence here.**

***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, REINFORCE, Proximal Policy Optimization (PPO), and Advantage Actor Critic (A2C). 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.
- 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

Deep Q learning uses a deep neural network to learn coefficients ω such that the network's value function evaluates $(state, action)$ pairs 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 state game state: the car position and car velocity.

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.

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

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

B. Double Deep Q Learning

Double deep Q networks address the maximization bias problem from Deep q learning by instead letting two Q functions randomly select the action and update the corresponding Q function. This process inhibits bias and will potentially converge faster or outright outperform DQN on all tasks.

C. 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 no direct affect on the game environment.

D. Multistep DQN Models

E. Basic Policy Gradient

Policy gradient algorithms in general are methods that learn a parameterized policy that can select actions without consulting a value function [5]. 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 [5]. 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.

F. 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 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 II-F.

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
```

```
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 {$\theta$}
```

G. 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. PPO also addresses other shortcomings of the Basic Policy Gradient by subtracting the state-value off 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.

H. 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 [5] 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 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.

Listing 2
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 3
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
```

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 2.

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 3.

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 1.

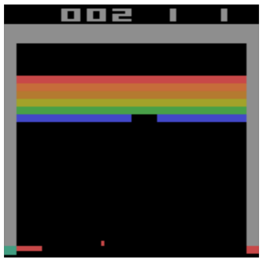

Fig. 1. **Example of the Breakout environment.** The Atari Breakout environment provided by openai.

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

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 2.

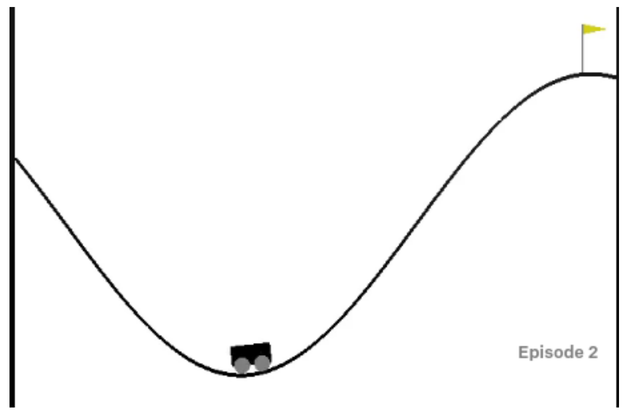

Fig. 2. **Example of the Mountain Car Environment** The mountain car environment provided by openai.

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.

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

- PUSH LEFT
- PUSH RIGHT
- NO PUSH

The state-space for this environment remains unchanged. 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 [5] 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.

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 4 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

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

Listing 4
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
```

E. 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.

The model structure and hyperparameters for this implementation can be seen in Listing 5 and Table II respectively.

TABLE II
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

Listing 5
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
```

IV. Results

A. Continuous Advantage Actor Critic

The continuous actor critic agent was exercised using the MountainCarContinuous-v0 environment. This agent as implemented 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 3. In this plot a scores near 80 are achieved for starting at episode 7000 with the highest score coming in at 91.06.

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 4, 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 5. In this plot one can see a pronounced general upward trend in the learning which has essentially converged around 9000 episodes.

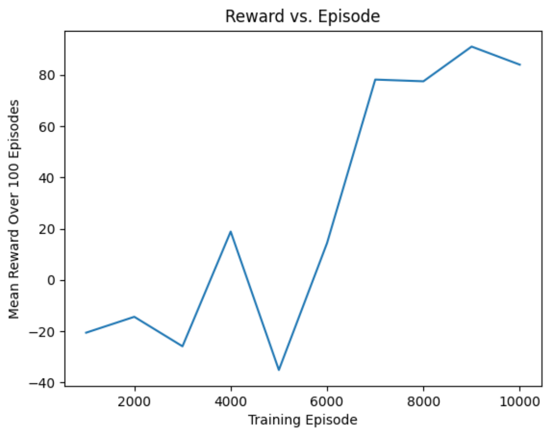

Fig. 3. **The actor critic agent playing MountainCarContinuous-v0** The actor critic agent successfully learns how to win the MountainCarContinuous-v0 environment.

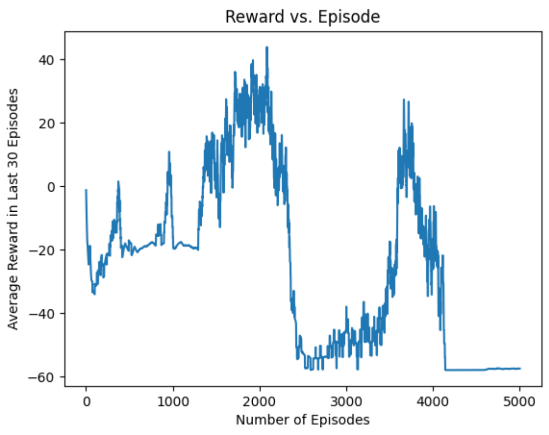

Fig. 4. **A bad training session for actor critic agent playing MountainCarContinuous-v0** The actor critic method has stability issues that can affect training.

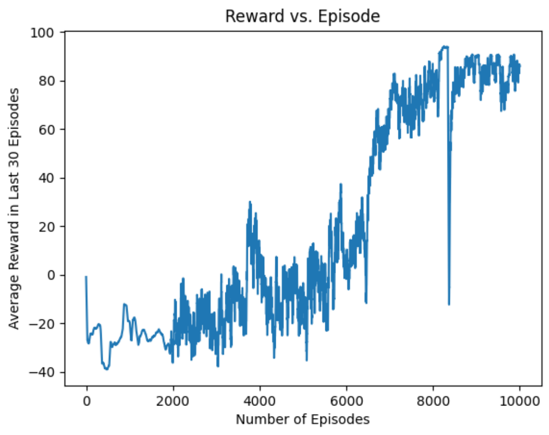

Fig. 5. **A typical training session for actor critic agent playing MountainCarContinuous-v0** A typical training session for the actor critic method.

B. 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 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 6.

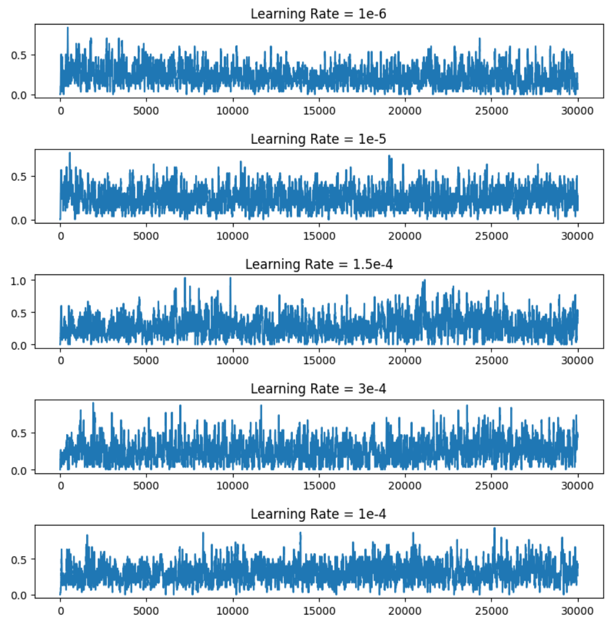

Fig. 6. **A search through various learning rates for actor critic on Breakout.** Changing the hyperparameter for learning rate did not improve the results for the discrete version of actor critic on the Breakout environment.

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 target network and 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.

V. Conclusion

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