Model-Free Deep Reinforcement Learning Algorithms in the MuJoCo Physics Simulator

Curtis Brinker
cjbzfd@mst.edu

Tanner May
tmay@mst.edu

***Abstract*—Deep reinforcement learning is a field that has shown rapid growth in the recent years. These advances have made progress in solving problems that were previously considered unapproachable such as the game Go or self driving vehicles. This paper looks to provide an introduction to deep reinforcement learning by examining several notable algorithms in detail. These algorithms are then implemented and tested in the MuJoCo physics simulation to provide quantitative and qualitative comparisons of the performance for each of the different algorithms.**

I. Introduction

Recent developments in machine learning have made huge strides in approaching problems that were previously unsolvable with traditional programming methods. In general, machine learning techniques are grouped into three categories: supervised learning, unsupervised learning, and reinforcement learning (RL). The first two categories train on a dataset with constant values. RL is unique in the fact that the data it trains on frequently changes due to the strong causal interdependency between the agent and the environment. As the training data is consistently changing, RL algorithms must be able to learn in real time, called online learning [1].

This paradigm of machine learning aligns well with several real life applications and as a result has led to new state-of-the-art algorithms to solve them. Some examples may include DeepMind's AlphaZero, which became the best Go player. Another example are RL algorithms that have achieved superhuman performance in a variety of Atari games using only visual inputs. However, RL algorithms can excel in more than just games, RL agents have been created to control robotics, design machine learning algorithms and much more [2].

RL is capable of learning on data as it comes in. This data is formulated as a set of interactions with the agent's environment where each interaction consists of the state of the environment,the action the agent performed, and the resulting state. Thus, agents that use RL learn with a circular process that is a result of the agent's previous actions, known as the agent-environment interaction loop. This complex relationship has given rise to multitudes of algorithms, from remembering a mapping between states and actions to using multiple neural networks to determine the next action .

In this paper we aim to give an introduction to the mathematics used in deep reinforcement learning by examining several famous deep RL algorithms. By learning the mathematics, we aim to learn the core principles that the field of RL uses, along with special tricks and techniques that are used to improve performance. Specifically, this paper looks to study and implement the following algorithms:

- REINFORCE
- A2C
- DDPG
- SAC

To test the performance of these algorithms, we trained models and recorded the improvement over time. The most intensive tests were performed in the MuJoCo simulator where agents were trained to walk in a 3D simulation. All code, models and figures from these tests are available at the project repo.[1]

II. Testing Environment

All algorithms were written and tested on a Linux system running Ubuntu 20.04. The system had the MuJoCo 2.1 installed according to the instructions provided in the project repo.[1] MuJoCo stands for **Mu**lti-**Jo**int dynamics with **Co**ntact and was made for applications that require fast, accurate physics simulations such as robotics and machine learning [3]. Using MuJoCo, machine learning algorithms can learn how to control the movement of robotic models, such as a snake, ant, or humanoid.

All RL algorithms were written using Python 3.8 and the PyTorch library. The algorithms were written using the OpenAI `gym` interface. This interface was used as it is a standardized way to interact with RL algorithms and the `gym` library has native support for MuJoCo. This interface is described in the documentation [4]. An overview of this interface is given below:

- `reset()` : Resets the environment to an initial state
- `step()` : Used to update the environment and returns the following:
  - An observation of the environment
  - The reward from the most recent step
  - A done flag, marking a state as a terminal state
  - An info object which has a log of the internal details of the environment step

III. Background

The study of reinforcement learning is about training an *agent* to interact with an *environment*. An action by an agent

[1] https://github.com/cubrink/mujoco-2.1-rl-project

can influence the environment, which may later affect the actions of the agent. For this reason, reinforcement learning algorithms must be able handle a changing environment that is causally influenced by the agent itself.

Formalizing the interdependency between the environment and the agent is done with the *agent-environment interaction loop*. In which, at time t the environment is fully described by its *state*, s_t. Then, an agent makes an *observation* of the environment, o_t, where $o_t \subseteq s_t$.[2] The agent responds to this observation with an *action* a_t and is given a *reward*, r_t. After the action is taken, the state of the environment changes with the new state denoted as s_{t+1}.

An environment state can be represented by a tensor of values describing individual aspects of the environment. The domain of possible observations of the agent is referred to as the *observation space*, which can be either continuous or discrete. Similarly, the domain of all actions that an agent can take is called the *action space* which can also be continuous or discrete. An action, a_t, is selected from the action space by agent's *policy*. The policy can select actions either deterministically or stochastically. Typically, deterministic policies are denoted with $a_t = \mu(s_t)$ and stochastic policies are denoted as $a_t \sim \pi(\cdot|s_t)$ [5].

The reward given for the transition from s_t to s_{t+1} by action a_t is given by the reward function, where $r_t = R(s_t, a_t, s_{t+1})$. Though, readers should be aware that some literature refers to this value as r_{t+1}. Typically, this reward function is specified by a programmer in a way that rewards a specific goal. With this in mind, the process of training an agent can be formulated as maximizing the cumulative reward, called the *return*. To calculate the return, the *trajectory* is used which contains the sequence of states and actions taken by the agent. The trajectory is given by:

$$\tau = (s_0, a_0, s_1, a_1, ...)$$

Therefore, as the return is defined in terms of cumulative reward, the trajectory can be used to calculate the rewards as the agent interacts with the environment. Then, in its most simple form, the return is the summation of all rewards. However this introduces two problems: First the return for an action t is defined in terms of actions taken before it. Intuitively, we would to only consider rewards that will happen as a result of the current action. Second, The summation of all rewards weights all rewards equally, however, this means that very long term rewards may eventually dominate the summation. To encourage shorter term rewards we can introduce a discount factor, γ, that discounts the value of future rewards.

Commonly, the reward function R is co-opted to take a trajectory and produce the rewards at each timestep t. With these considerations in mind, the return can be defined as:

$$R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'} r_{t'}, \;\; \gamma \in (0, 1)$$

[2]While the observation and the state are not necessarily equal, reinforcement learning literature frequently refers to an observation as the state itself.

Where T is the length of the trajectory and t is the current time for which the return is being calculated. If T is finite, then the return is said to be a *finite-horizon return*. Similarly, if T is infinite, then the return is said to an *infinite-horizon return*. The inclusion of γ makes this return a discounted return. This places more focus on near-term rewards and also helps with convergence in infinite-horizon returns. Using t' makes the return only consider future rewards, which [6] calls the *reward-to-go* return.

However, an agent cannot act to directly maximize its return as it dependent on future actions and states. Instead, an approximation must be used to calculate the expected return, $\mathcal{J}(\pi)$. This this done with *value functions*. Formally, $\mathcal{J}(\pi)$ is defined as:

$$\mathcal{J}(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]$$

The *state value function* for policy π, denoted as V^π, gives the expected value for policy π given state s. Alternatively, the *state-action value function* for policy π, denoted as Q^π, which can be used estimate the expected value for policy π in state s and immediately taking action a. Formally, these are defined as:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau) \,|\, s_0 = s]$$
$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau) \,|\, s_0 = s, a_0 = a]$$

However, these functions are still dependent on the return. To avoid this, we can use the Bellman Equations, which defines both V^π and Q^π in terms of expected values of future states. [2], [5], [7] These equations are given as:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim P}}[R(s, a, s') + \gamma V^\pi(s')]$$
$$\begin{aligned} Q^\pi(s, a) &= \mathop{\mathbb{E}}_{s'\sim P}\left[R(s, a, s') + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}[Q^\pi(s', a')]\right] \\ &= \mathop{\mathbb{E}}_{s'\sim P}[R(s, a, s') + \gamma V^\pi(s')] \end{aligned}$$

Where P is the environment state transition function and s' is the resulting state.

For the agent to maximize $\mathcal{J}(\pi_\theta)$, the agent must learn to improve the policy π_θ, where θ are the parameters for the policy and is normally updated by gradient ascent. Normally this involves using V^π or Q^π to better approximate the optimal value functions. Commonly, these approximation are done via deep neural networks. The exact details of how the policy and value function(s) are updated is dependent on the reinforcement learning algorithm used. Following is a brief discussion on different types of reinforcement learning algorithms followed by discussions on details of specific algorithms studied in this paper.

A simplified taxonomy of deep reinforcement learning algorithms is given by [8]. At the highest level, reinforcement learning algorithms can be split into two categories, *model-based* or *model-free*. Model-based algorithms learn the state transition function, allowing it to predict future states of the environment. Learning the world model has many benefits

that can drastically increase the performance of an agent, however, doing so significantly increases the complexity of the algorithms and is outside the scope of this paper. Model-free algorithms do not learn to predict the environment and instead learn other functions to increase the agent's return. Model-free algorithms can be further categorized as using *policy optimization* or *Q-learning*.[3]

Policy optimization algorithms train by optimizing the parameters of the policy π_θ. This is normally done learning a value approximator V_ϕ and sampling actions taken by the current policy to update parameters accordingly. Because this process requires actions from the current policy, these techniques are considered to be *on-policy*.

Another approach, though not necessarily mutually exclusive, is using Q learning. These algorithms learn an approximator Q_θ to determine the value of state action pairs. An advantage to this technique is that it can learn from past experiences that were not from the current policy. Algorithms that can learn from experiences of different policies are called *off-policy*. Actions by these algorithms typically are chosen by selecting the action that maximizes $Q_\theta(s, a)$ for the current state s.

A. The REINFORCE Algorithm

One of the simplest forms of policy optimization algorithms is a class of algorithms presented by Williams as REINFORCE [9]. The core idea this algorithm is to update the policy by sampling trajectories from the policy. Once a sample has been taken from the current policy, the policy π_θ and value approximator V_ϕ can be updated. There is some flexibility in how the parameters are updated, though [6] provides some examples of frequently used techniques. The policy is updated using gradient ascent where the gradient is given by:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \, log \, \pi_\theta(a_t|s_t) \, \Phi_t \right]$$

Where Φ_t is any function that is independent of θ.

To update the value approximator parameters ϕ, it is common to use gradient descent on a loss such as mean squared error on the value approximator and the reward-to-go return.

$$\mathcal{L}(\phi) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\left(V_\phi(s_t) - R_t(\tau) \right)^2 \right]$$

B. Advantage Actor Critic (A2C)

The advantage actor critic is an on-policy algorithm based on the REINFORCE algorithm. A2C was first proposed by [10] along with an asynchronous variant called A3C. The details about the asynchronous version of the algorithm is not pertinent to this paper and will not be discussed. As a whole, there are two main techniques used to improve the performance of this algorithm.

The first technique is the use of the advantage function, $A^{\pi_\theta}(s_t, a_t)$ as Φ_t. The advantage function is defined as:

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t, a_t)$$

[3]Policy optimization and Q-learning are not mutually exclusive categories.

It should be noted that Φ_t must be independent of θ and by consequence a_t. However, [6] provides a proof that $A^{\pi_\theta}(s_t, a_t)$ can be written it terms of s_t only. The advantage can be thought of as quantifying how much better an action in a given state is than average. Negative values signify that the action was less than the expected value for the given state, and positive signifies that it was better. Mathematically, the advantage function is useful as subtracting by $V^{\pi_\theta}(s_t)$ reduces the variance in the policy gradient leading to more stable learning.

The second technique is the use of an actor-critic framework. The idea of this framework it to separate the concerns of learning different aspects of training. The actor is tasked with learning the policy that will interact with the environment. The critic is tasked with learning the value function that it will use to criticize the actor's actions [11].

C. Deep Deterministic Policy Gradient (DDPG)

DDPG is an off-policy learning algorithm for continuous action spaces that was proposed by [12]. It learns a Q function and a deterministic policy to select actions and uses a variety of techniques to improve performance and stability of the algorithm. An overview of unique methods used by this algorithm is provided by [13]. One technique that is commonly used by off-policy algorithms is the use a replay buffer. Because off-policy algorithms can learn from actions that did not originate from the current policy, past experiences can be stored in a buffer. Then, during training, a larger sample size can be used to improve training.

To update the function Q_ϕ, gradient descent is used on ϕ to minimize the mean squared error of $Q_\phi(s, a)$ and the *target*, where the target is defined as:

$$y = r + \gamma(1 - d) Q_\phi(s', \mu_\theta(s'))$$

Where (s, a, r, s', d) is a transition in the replay buffer $\mathcal{D}$. The resulting state from the action is s' and d is the *done* flag where $d = 1$ if s' is a terminal state, $d = 0$ otherwise. Additionally, μ_θ is the deterministic policy used to select the next action.

However, using gradient descent on the mean squared error of Q_ϕ and y is problematic as y is also dependent on ϕ. As a result, both Q_ϕ and y change when ϕ is updated, leading to instability. To resolve this, DDPG uses *target networks*, which uses polyak averaging to remove this direct dependency. The target networks, $Q_{\phi'}$ and $\mu_{\theta'}$ have the same structure as their original networks but are updated as follows:

$$\phi' \leftarrow \tau \, \phi' + (1 - \tau)\phi$$
$$\theta' \leftarrow \tau \, \theta' + (1 - \tau)\theta$$

For $\tau \in (0, 1)$. This provides stability to the loss function, therefore Q_ϕ is updated using gradient descent on

$$\mathcal{L}(\phi, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_\phi(s, a) - y \right)^2 \right]$$

This loss function is called the Mean Square Bellmen Error (MSBE) where y is the target, which is calculated as:

$$y = r + \gamma(1-d)Q_{\phi'}(s', \mu_{\theta'}(s'))$$

To update the policy, we want to maximize the value function Q_ϕ using μ_θ. To do this, the following loss function is used:

$$\mathcal{L}(\theta, \mathcal{D}) = \underset{s\sim\mathcal{D}}{\mathbb{E}}\left[-Q_\phi(s, \mu_\theta(s))\right]$$

Finally, to gain additional benefit from the algorithm off-policy, DDPG uses another technique to encourage exploration during train time. This is done by adding random noise to actions to stray away from the deterministic policy. The original authors uses OU noise, however [13] recommends a zero centered Gaussian distribution which is simpler to implement and still effective.

D. Soft Actor Critic (SAC)

The Soft Actor Critic Algorithm is an stochastic off-policy algorithm that uses an actor-critic framework and entropy regularization. This algorithm was originally proposed by [14], [15], additionally, a guided explanation is given by [16]. SAC can be implemented for either discrete or continuous action spaces, however, the details that follow will be tailored for the continuous version.

SAC works by learning a stochastic policy π_θ and two Q functions, Q_{ϕ_1} and Q_{ϕ_2}. When evaluating the value, the minimum value is taken between $Q_{\phi_1}(s,a)$ and $Q_{\phi_2}(s,a)$. This is called the *clipped double-Q method*. Using this method helps prevent overestimating the value of state action pairs which improves training results.

Similar to DDPG, SAC uses target networks to stabilize the updates using the loss function. Unlike DDPG, SAC only uses target networks for value functions, not for policy. The parameters for the target networks are updated using polyak averaging as shown below:

$$\phi_1' \leftarrow \tau\,\phi_1' + (1-\tau)\phi_1$$
$$\phi_2' \leftarrow \tau\,\phi_2' + (1-\tau)\phi_2$$

For $\tau \in (0,1)$.

To control exploration of the policy, SAC uses entropy regularization, where entropy is defined as:

$$H(P) = \underset{x\sim P}{\mathbb{E}}\left[-log\,P(x)\right]$$

For any distribution P. Roughly speaking, entropy measures how random a distribution is. An entropy of zero is given by a policy where where an action has probability 1. Larger values of entropy indicate more evenly spread probability distribution. The choice of entropy regularization is controlled with the entropy hyperparameter, α, where $\alpha \in (0,1)$. The greater the value of α, the more the policy will explore.

Entropy is introduced in several aspects of the SAC algorithm. The first is in the *entropy regularized double-Q clipped target*, which is calculated as:

$$y = r + \gamma(1-d)\left(\underset{k=1,2}{min}\, Q_{\phi_k'}(s', a') - \alpha\, log\, \pi_\theta(a' \mid s')\right)$$

Where $a' \sim \pi_\theta$.

This is used to calculate the entropy regularized MSBE which is used as the loss function for Q_{ϕ_1} and Q_{ϕ_2}, given by:

$$\mathcal{L}(\phi_k, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}}\left[\left(Q_{\phi_k}(s,a) - y\right)^2\right],\ \ k = 1,2$$

Where $\mathcal{D}$ is the experience replay buffer.

The policy π_θ is updated in a way similar to DDPG. The policy is updated by using gradient ascent to maximize the value function. Therefore the loss function for π_θ becomes:

$$\mathcal{L}(\theta, \mathcal{D}) = \underset{\substack{s\sim\mathcal{D}\\ a\sim\pi_\theta}}{\mathbb{E}}\left[-\left(\underset{k=1,2}{min}\, Q_{\phi_k'}(s,a) - \alpha\, log\, \pi_\theta(a \mid s)\right)\right]$$

There are two important details regarding sampling action a from π_θ in the SAC algorithm. First, sampling is done from a normal distribution using the *reparameterization method*. This makes the randomly sampled number differentiable, allowing for actions to backpropagate through the policy's parameters. The second detail is that the SAC policy squashes the action into a finite action space, with $tanh$. This is done as most action spaces have finite bounds while normal distributions have infinite support. By limiting the domain of the action to the action space, the policy is learned more efficiently.

IV. Methodology

Testing of the effectiveness of RL algorithms was done sequentially and in multiple stages, increasing in difficulty. The first tests were algorithms that used a discrete action spaces, REINFORCE and A2C. The tests were performed in the `CartPole-v0` environment. These tests were qualitative in nature, with the goal seeing that the simplest algorithms were properly learning.

The second group of tests focused on testing the implementation of algorithms with a continuous action spaces, DDPG and SAC. The tests were performed in the `Pendulum-v1` environment. These tests were quantitative in nature and recorded the improvement of the agent over time.

The third group of tests focused on testing the performance of algorithms with a continuous action spaces. The algorithms tested were A2C, DDPG and SAC. The tests were performed in the MuJoCo `Ant-v3` environment. These tests were quantitative in nature and recorded the improvement of the agent over time.

The fourth test took the best performing algorithm from the third group of tests and subjected it to a more challenging environment, MuJoCo's `Humanoid-v3` environment. This environment has a larger observation and action space than the

`Ant-v3` environment. This test was quantitative in nature and recorded the improvement of the agent over time.

The hyperparameters for each algorithm were as follows:

- REINFORCE: Train for a maximum of 5000 episodes. Update at end of each episode. Learning rate of 3e-4.
- A2C: Learning rate of 3e-4 and $\gamma = 0.99$.
- DDPG: Update freq. of 64 steps, update threshold of 4096 steps, batch size of 128, learning rate of 1e-3, $\gamma = 0.99$, $\tau = 0.995$, and the noise distribution was Gaussian with a standard deviation of 0.1.[4]
- SAC: Update freq. of 64 steps, 64 updates per update step, an update threshold of 4096 steps, batch size of 128, $\alpha = 0.5$, learning rate of 5e-4, $\gamma = 0.99$, $\tau = 0.995$.

Each of those hyperparameters were chosen based roughly on existing literature. Values were tweaked until they resulted in a model that was capable of learning the environment.

All deep neural networks used in these algorithms were multilayer perceptrons with two hidden layers of size 256 each, using the ReLU activation function. From our testing, using 128 neurons per layer frequently caused problems when learning the environment. At 256 neurons, all models seemed to be able to learn the environment sufficiently and adding more neurons would have greatly increased the computational cost of training.

More detailed information about the tests in each environment is described in its own subsection below the source code for each environment is given by [17].

A. CartPole-v0

The objective of the CartPole environment is to teach an agent to balance a pole on a cart by moving the cart left and right. The environment has a continuous observation space consisting of the cart's position, the cart's velocity, the pole's angle, and the pole's angular velocity. The action space is discrete with the options of moving the cart to the left or to the right. The environment rewards the agent with a value of one, simply for staying healthy.

When the environment is initialized the values of both the cart and pole are randomly sampled from a uniform distribution from $[-0.5, 0.5]$. The environment is terminated if: the angle of the pole is $\pm 12^\circ$ off center, the cart position hits the edge of the display,or when the length of the episode surpasses 200 steps. The environment is considered solved when the reward is$>$195 for 100 consecutive episodes.

This environment was used as a qualitative test for the implementation of algorithms with a discrete action space.

B. Pendulum-v1

The pendulum environment's objective is to balance a frictionless pendulum straight up. The observation space is continuous and describes the angle of the pendulum as well as the angular velocity. The action space is also continuous and details the amount of left or right force to apply to the pendulum. The reward incentivizes keeping the pole at an angle of 0° with as little velocity and force as possible. The reward is formulated as

$$R = -(angle^2 + 0.1 * angular\_velocity^2 + 0.001 * action^2)$$

Since the angle is normalized between $[-\pi, \pi]$ before calculating the reward, the reward has a range of$[-16.3, 0]$.

The environment's initial state has the pole at a random angle between $[-\pi, \pi]$ radians with a random velocity between $[-1, 1]$. The environment does not specify a termination state, so a limit of 150 steps was imposed. Similarly, the environment does not specify when it is solved, so it was allowed to run until it hit the step limit.

Similar to the CartPole environment, this environment was used to test the implementation of algorithms with continuous action spaces.

C. Ant-v3

The MuJoCo Ant-v3 environment has the goal of teaching an ant to move as quickly as possible to the right. The ant is a sphere with four legs, each with two joints. The continuous observation space is significantly more complex than the previous environments: it consists of the model's position, rotations, velocity, and the forces between the legs and the ground (contact force). The action space is also continuous and describes where and how quickly to move each joint. The reward encourages the model to move as quickly as possible while moving as few joints as little and softly as possible. It is formulated as

$$R = (v + h) - (a + f)$$

where v is the velocity in the x-axis, h is the healthy reward configured to be 1, a is the control_cost calculated by $0.5 * \sum actions^2$, and f is the contact_cost calculated by 5e-4 $* \sum contact\_forces^2$.

The environment starts with each joint in a random position with a random velocity. The episode is terminated when the model exits the safe height range of $[0.3, 2.0]$. The environment did not define a solve condition so the episode was continued until the model made a mistake to trigger the termination condition.

The ant was chosen because it is the simplest provided model that meets the "three dimensional walker" condition. Each of the algorithms had 500,000 episodes to train using this model. After training was stopped, the most recently saved model was used for analysis. A training length of 500,000 episodes was used because it consistently produced an agent that could walk a non-trivial distance.

D. Humanoid-v3

The MuJoCo Humanoid-v3 environment has the goal of teaching an humanoid to move as quickly as possible to the right. The humanoid model is a humanoid with two legs, two arms, a head and a torso. The legs, arms, and torso are made up of multiple joints, each individually controlled. Like the ant, the observation space is continuous and describes the

[4] [12] uses OU noise, but [13] recommends using Gaussian noise as it is easier to implement and gives similar results

model's position, the model's velocity, the forces for each of the joints, and the contact forces. The environment also defines a continuous action space that describes the new position and velocity of each joint. The intuition for the reward is the same as for the Ant environment, but it is calculated slightly differently where v is equal to $1.25 * x\_velocity$, h is configured to be 5, a is calculated as $0.1 * \sum actions^2$, and f is defined as 5e-7 $* \sum contact\_forces^2$.

Like the Ant environment, the model starts with each joint in a random position and random velocity. The episode is terminated when the model exits the safe height range of $[1.0, 2.0]$. The environment did not define a solve condition so the episode was continued until the model made a mistake to trigger the termination condition.

The humanoid was chosen because it is the most complex 3D walker and showcased the learning power of the tested algorithms. Since this model is significantly more complex than the Ant, 1,250,000 environment interactions were used for training. But due to the cost of training, only SAC was trained with this model.

V. Results

The following sections discuss each algorithms' results for each of the environments, including graphs of the reward history and various loss values. As [5] notes, unlike other machine learning techniques, supervised learning for example, the loss function is defined on the most recent policy, so it is only relevant for calculating the descent gradient *for that specific version of the policy*. This means that minimizing the loss function is not guaranteed to improve the expected return, and may in fact do the opposite. Essentially, for RL the loss values mean nothing, but their graphs were included here for posterity's sake.

To see videos of the agents' performance in some of the following environments see [18].

A. CartPole-v0

The CartPole test was qualitative in nature and was used to provide feedback that the our earliest implementations of the algorithms were working. Both the REINFORCE and A2C algorithms were able to solve the environment in a matter of a few minutes. Qualitatively the A2C appeared to have better performance in this environment.

B. Pendulum-v1

The tests on the pendulum environment proved that our implementations of A2C, DDPG and SAC were able to solve problems with continuous action spaces. Looking at the reward history in the Figures 1 and 2, it appears that DDPG was able to solve the algorithm faster than SAC. This isn't unexpected considering that SAC uses significantly more neural networks than DDPG and may take longer to train.

C. Ant-v3

While each of the models were able to move the Ant model, they differed in the degree of "intelligence" they showed. Both

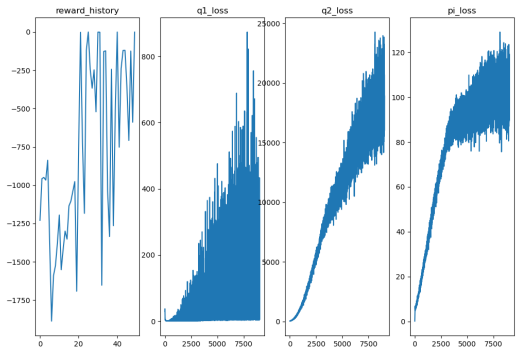

Fig. 1: SAC in the Pendulum-v1 Environment

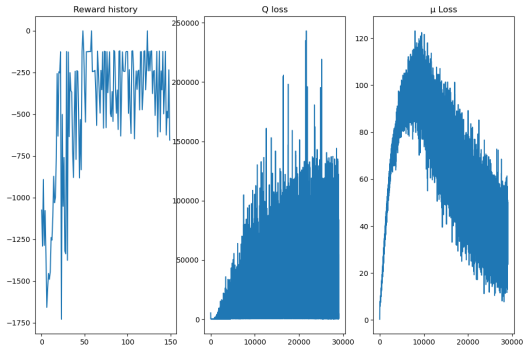

Fig. 2: DDPG in the Pendulum-v1 Environment

A2C and DDPG learned to move the ant to the right, but their techniques were inconsistent and sporadic. Both methods frequently caused the ant to jump and flip around in the generally correct direction, but also frequently would get stuck upside down shortly after beginning.

DDPG was able to use the left and right legs to walk, and used the front and rear legs for steering. Using this method it was able to achieve quite the distance. But SAC managed to

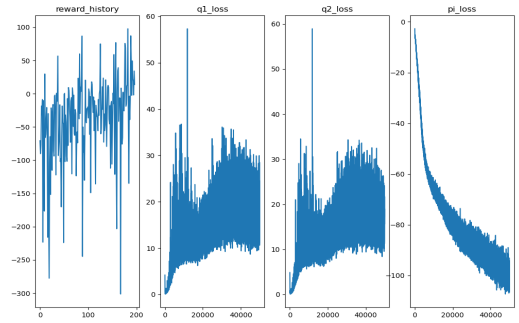

Fig. 3: SAC in the Ant-v3 environment

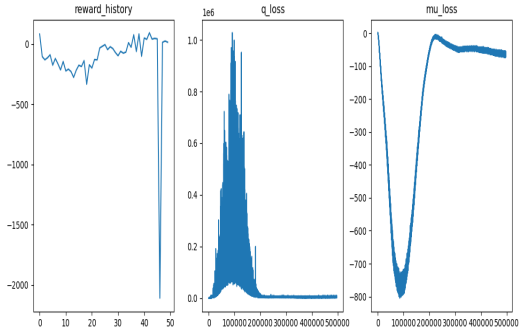

Fig. 4: DDPG in the Ant-v3 environment

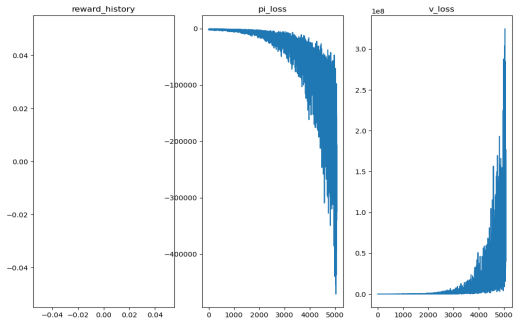

Fig. 5: DDPG in the Ant-v3 environment

top even that; using the same locomotion scheme as DDPG, the model learned to leap through the air and even reached the edge of the floor in the environment. This is reflected in the performance graphs: the reward history for SAC increases at a steeper rate than it does for DDPG.

The main result from this test is that while all algorithms were able to make progress in the environment, SAC was able to significantly outperform the other algorithms. It learned to control the ant in a way that kept itself stable physically and move efficiently.

D. Humanoid-v3

Since SAC's performance on the Ant model was so impressive, it was the first to train with the Humanoid model. As shown in the performance graph, it quickly learned how to walk using the arms for balance. After that initial hurdle, it refined its balance technique an opted for a slow and steady strategy by shuffling the feet along the floor. *The SAC Shuffle* allowed the model to walk far into the distance, effectively solving the environment.

Unfortunately, the Humanoid-v3 environment was the most complex environment tested. The SAC model trained for 1,250,000 environment interactions which took over 12 hours using a NVIDIA RTX 3060 Ti. For this reason, testing algorithms we believed to be worse than SAC could not be justified due to time constraints.

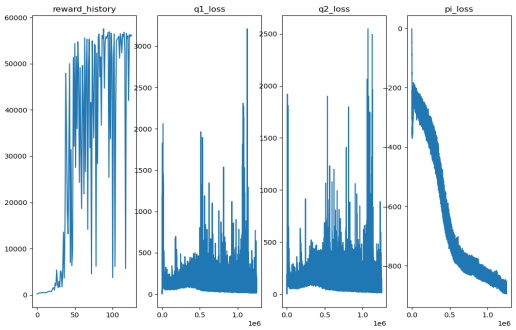

Fig. 6: SAC in the Humanoid-v3 environment

VI. Conclusions and Future Work

Learning to walk is a challenging problem; multiple limbs have to move in a coordinated fashion just to keep balance, adding movement on top of that is a problem that is potentially unsolvable with conventional programming methods. Using Reinforcement learning algorithms we were able to teach an agent to move a complex model in a 3D environment. Further, we found that some models such as SAC were able to learn complex behaviors such as leaping to improve its performance.

By implementing and training these algorithms, we are also leaping into success: we now have a solid foundation to begin building new experiences in Reinforcement Learning. We were introduced the foundations of RL like the value functions and policies, policy gradients, and a dictionary's worth of RL vocabulary. This knowledge of RL theory and application and is undoubtedly valuable for future projects in research and industry.

This project involving RL with MuJoCo could be continued in several different ways. One continuation we can explore would be implementing more algorithms or train with different kinds of environments that impose other interesting challenges. For example, can SAC learn to ride a bicycle? Another way to continue this project would be to explore the affects of hyperparameters in RL algorithms. Do certain hyperparameters affect training similarly between algorithms? How might they differ between algorithms? What are the optimal hyperparameters for different types of environments?

However, reinforcement learning has a much broader scope than just MuJoCo. The algorithms that we implemented could move on to Multiagent Learning for games like Chess [19], learning how to play complex single player video games like Dark Souls, or even to control self driving vehicles [20]. Our new foundational knowledge of RL leaves us well equipped to keep up with research and contribute to the body of knowledge that is continuing to grow daily.

References

[1] W. Qiang and Z. Zhongli, "Reinforcement learning model, algorithms and its application," in *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, 2011, pp. 1143–1146.

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[3] DeepMind. Mujoco documentation. [Online]. Available: https://mujoco.readthedocs.io/en/latest/overview.html

[4] OpenAI. Gym documentation. [Online]. Available: https://gym.openai.com/docs/

[5] (2018) Spinning up: Key concepts in rl. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

[6] (2018) Spinning up: Intro to policy optimization. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[8] (2018) Spinning up: Kinds of rl algorithms. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[9] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, p. 229–256, may 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[11] H. van Hasselt. (2021) Reinforcement learning lecture 9: Policy gradients and actor critics. [Online]. Available: https://storage.googleapis.com/deepmind-media/UCL%20x%20DeepMind%202021/Lecture%209-%20Policy%20gradients%20and%20actor%20critics.pdf

[12] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[13] (2018) Spinning up: Deep deterministic policy gradient. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ddpg.html#

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018. [Online]. Available: http://arxiv.org/abs/1801.01290

[15] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," *CoRR*, vol. abs/1812.05905, 2018. [Online]. Available: http://arxiv.org/abs/1812.05905

[16] (2018) Spinning up: Soft actor-critic. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/sac.html

[17] OpenAI. Gym library. v0.21.0. [Online]. Available: https://github.com/openai/gym

[18] C. Brinker and T. May. mujoco-2.1-rl-project. [Online]. Available: https://github.com/cubrink/mujoco-2.1-rl-project/tree/main/experiments

[19] S. Albrecht and P. Stone. (2017) Multiagent learning: Foundations and recent trends. [Online]. Available: https://www.cs.utexas.edu/~larg/ijcai17_tutorial/multiagent_learning.pdf

[20] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Perez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 99, pp. 1–18, 2021.