National College of Ireland

# Developing an Artificial Agent to play Games using Deep Reinforcement Learning

MSc Research Project
Data Analytics

Chetan Bhardwaj
Student ID: 20176724

School of Computing
National College of Ireland

Supervisor: Dr. Mohammed Hasanuzzaman

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

National College of Ireland

| | |
|---|---|
| **Student Name:** | Chetan Bhardwaj |
| **Student ID:** | 20176724 |
| **Programme:** | Data Analytics |
| **Year:** | 2021 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Dr. Mohammed Hasanuzzaman |
| **Submission Due Date:** | 16/12/2021 |
| **Project Title:** | Developing an Artificial Agent to play Games using Deep Reinforcement Learning |
| **Word Count:** | 6017 |
| **Page Count:** | 20 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | Chetan Bhardwaj |
| **Date:** | 30th January 2022 |

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:**

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies). | ☐ |
| **Attach a Moodle submission receipt of the online project submission**, to each project (including multiple copies). | ☐ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | ☐ |

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Developing an Artificial Agent to play Games using Deep Reinforcement Learning

Chetan Bhardwaj
20176724

**Abstract**

In Reinforcement Learning, Deep Q-Learning agents have seen great success with popular agents such as Alpha GO and Alpha Zero, these agents are constrained to discrete state space environments and are even prone to overestimation bias. To address these restrictions, we deployed Deep Deterministic Policy gradient and Soft Actor Critic off-policy algorithms in the Lunar Lander environment described in Zhao et al. (2020) research and compared their performance to Deep Q-Networks. We observed that out of the three implemented algorithms, SAC outperformed both DQN and DDPG algorithms with a huge margin.

# 1 Introduction

## 1.1 Motivation and Background

The gaming industry, which is approximately worth $159 billion,[1] is one of the largest in the entertainment industry. An ever-expanding sector that seeks to provide something exquisite and intriguing to the world's 2.7 billion players, each game unique and superior to the others, encouraging creators to seek out more complex and novel game structures that can compete in this competitive environment. Developers are continually attempting to create agents capable of rivalling human strategic tendencies and assisting players in Player vs Player (PvP) matches. This investigation led the developers to Reinforcement Learning, a deep learning technique.

"Reinforcement learning is aimed at controlling a computer agent so that a target task is achieved in an unknown environment" Sugiyama (2015). A simple but effective method of accomplishing this goal is through trial and error. Each goal achieved by the AI is rewarded, while each error is punished by a negative score or a penalty.

[1] https://techjury.net/blog/video-games-industry-statistics/#gref

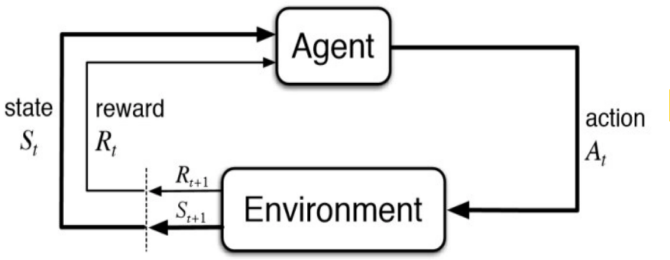


Figure 1: Interaction of RL agent with environment

The figure above explains the interaction of a RL agent in the environment. The agent takes an action in the environment in turn the environment rewards the agent and gives a feedback of the updated state of the environment.

The applicability of reinforcement learning models is strongly dependent on the environment in which they are deployed, and building such environment is a complex and time-consuming process. The environments are incredibly dynamic and vary according to the problem statement. For example, the simulation environment for an autonomous car is considerably different from the simulation environment for a gaming simulation. For this research, we will focus on the gaming environment.

In this study, we propose to train a Deep Q-Network (DQN) RL agent in the Lunar Lander Environment using the OpenAI gym library, which was used by Zhao et al. (2020) in their research to construct agents which could aid developers during game development. We also compare its performance to that of newer, state-of-the-art algorithms such as Soft Actor Critic and Deep Deterministic Policy Gradient.

### 1.2 Research Question and Objectives

"How can we overcome the shortcomings of the Deep Q-Networks method by employing the Deep Deterministic Policy Gradient and Soft Actor Critic algorithms in the environment exploited by Zhao et al. (2020) in Lunar Lander?"

The remainder of the paper is organized as follows. The next part digs into earlier research on DQN's accomplishments and limits, as well as a discussion of both the DDPG and SAC algorithms. Sections 3 and 4 of the paper present the environment as well as the technique for deploying agents in the environment. Sections 5 and 6 of the paper offer a description of the project's implementation, evaluation methodologies, and outcomes. Section 7 closes the work by providing an overall overview as well as recommendations for additional research.

## 2 Related Work

This section's topic is linked to the previous reinforcement learning literature related to their implementation in the gaming domain. It is divided into four sections: 1) Deep Reinforcement Learning in Games using Deep Q-Networks 2) Reinforcement Learning with DDPG and SAC 3) Comparison and summary.

## 2.1 Deep RL in Games using Deep Q-Networks

The development of reinforcement learning agents began with TD-gammon in 1995, when an agent trained through self-play achieved superhuman level performance in the backgammon game (Tesauro (1995)). In addition to game-playing agents, another technique is Imitation learning which was investigated by Gorman and Humphrys (2007) . In the conducted research,the agent was trained using features extracted from an extensive dataset gathered from the gameplay of actual players of the game Quake 2, and the agent was assessed on the basis of its ability to play the game comparable to that of a human player.

However, the research did not take off in the gaming domain until 2013, when Bellemare et al. (2013) developed a 2600 Atari emulator for reinforcement learning and Hausknecht et al. (2014) contributed to this emulator by using HyperNEAT architecture, which further evolved the emulator's opening. Mnih et al. (2013) developed the world's first deep learning model capable of learning policies straight from raw sensory inputs from environments. This model uses a Q-learning variation, stochastic gradient descent, to update the weight and a memory replay method to randomly sample past events and improve training over time. The Atari 2600 provided a complex environment for the agent, and despite not being given any game-related information, the agent excelled any previous implementation and surpassed human level in three of those games. Another feat this state of art model achieved was no architectural or hyperparameter change was made to the model before deploying in different environments. Finally, in 2015 Mnih et al. (2015) developed novel Deep Q-Network an upgrade from the previous version and used the algorithm to train on 49 of those Atari games. The research by Jaderberg et al. (2019) lauds the improvements in the development of RL agents but notes that because games are no longer 2D, agents should experience more complex settings, necessitating the need to extract additional features from the environment. The researchers successfully implemented RL on the Quake 3 game, this 3D game that is similar to real life.

Reinforcement learning models seek all feasible ways to gain greater and greater rewards, revealing previously unknown pathways. Silver et al. (2018) presented Alpha Zero, a newer, better and a generic variant of Alpha Go that mastered Go, chess and Shogi. Alpha Go and its suc-cessor The difference between the two was: Alpha-Go calculates the winning probability of each possible move before making one while Alpha-Zero Go estimated and optimized the expected outcome of the move.

Researchers are increasingly focusing their efforts on generic RL agents, which should be able to operate in a variety of situations without requiring significant changes in hyperparameters. DeepMind's Alpha Zero is one such example. Researchers such as de Almeida and Thielo (2020) and Zhao et al. (2020) have effectively trained agents to earn rewards in a variety of environments. Zhao et al. (2020) conducted research on the development of a generic agent to facilitate game development during the testing phase and even realistic participants in a PvP fight, which is similar to the goal of this suggested research.

Jäger et al. (2021) addresses some crucial limitations of DQN networks like Q-learning is prone to selecting overestimated actions and adopting overoptimistic value estimations because it uses the same estimate for action selection via the max operator as well as

for evaluating this action, which makes DQN affected by the problem of overestimation bias. Q-learning is prone to selecting overestimated actions and adopting overoptimistic value estimations because it employs the same estimate for action selection via the max operator as well as for evaluating this action Van Hasselt et al. (2016).

Since targets are computed based on subjective estimates, training frequently becomes unstable, manifesting as a decline in performance and forgetting of previously acquired tasks, a phenomenon known as catastrophic forgetting. This difficulty is produced by learning on correlated samples because the environment's transitions are sequentially experienced and the Q-network's parameters I indirectly predict the next samples, which is somewhat solved using experience replay. Shortcomings like these motivates us to explore more algorithms in Reinforcement Learning like Deep Deterministic Policy Gradient (DDPG) and Soft Actor Critic (SAC), off-policy algorithms like DQN.

## 2.2 Reinforcement Learning with DDPG and SAC

Lillicrap et al. (2015) asserts that while DQN is able to solve problems involving high-dimensional observations, it is limited to discrete and low-dimensional action spaces, whereas many real-world situations include continuous and high-dimensional action spaces. To deal with continuous space difficulties, DQN discretizes the action space first. However, the problem of dimensionality arises as the number of actions increases exponentially with increase in the number of degrees of freedom. Hence proposing a new algorithm named Deep Deterministic Policy Gradient (DDPG). DDPG is an approach for off-policy reinforcement learning that combines Q-learning and Policy gradients. As an actor-critic technique, DDPG has two models: Actor and Critic. Instead of a probability distribution of actions, the actor is a policy network that takes the state as input and outputs the exact action (continuous). The critic is a Q-value network that accepts state and action as input and returns the Q-value as output. The approach's simplicity is a significant feature: it requires only a fundamental actor-critic architecture and learning algorithm with few "moving parts," making it simple to deploy to more complex situations and larger networks.

The model-free approach, dubbed Deep DPG (DDPG), enabled the researchers to develop competitive policies for all tasks using low-dimensional observations (e.g. cartesian coordinates or joint angles), while maintaining the same hyper-parameters and network topology. They were also able to learn effective rules directly from pixels in many cases, while maintaining the hyperparameters and network structure unchanged. The proposed algorithm DDPG used fewer steps to achieve same level of experience than used by DQN for Atari Environment but still requires a large number of training steps to find solutions. Hence considering the amount of time consumed while training, a new and better algorithm was needed which could achieve better results in fewer training episodes.

Soft Actor Critic (SAC), a Reinforcement Learning model developed by Haarnoja et al. (2018), highlighted the problems with on-policy algorithms rigorously. The author discusses the issue of poor sampling efficiency in on-policy algorithms such as PPO (Schulman, Wolski, Dhariwal, Radford and Klimov (2017)) and A3C (Mnih et al. (2016)), claiming that samples must be gathered after every gradient step, which becomes increasingly expensive as task complexity increases. It also raises the issue of dealing with

high-dimensional problems using off-policy algorithms like DQN Mnih et al. (2015), as stability is reduced in such cases. For its extreme brittleness and sensitive hyperparameters, DDPG is difficult to train Henderson et al. (2018). SAC was designed to address challenges in continuous state and action spaces, resulting in a stable method that also delivers sample-efficient learning.

Three critical components comprise the soft actor-critic algorithm: an actor-critic architecture with distinct policy and value function networks, an off-policy formulation that allows efficient reuse of previously gathered data, and entropy maximization for stability and exploration.Maximum entropy reinforcement learning improves policies to maximize both anticipated return and expected entropy. The maximum entropy distribution is utilized in guided policy search Levine and Koltun (2013) to drive policy learning towards high-reward regions. Several articles have lately emphasized the relationship between Q-learning and policy gradient approaches in the context of maximum entropy learning Schulman, Chen and Abbeel (2017).

In terms of sample efficiency, the SAC algorithm outperformed the DDPG algorithm, demonstrating that stochastic, entropy-maximizing reinforcement learning algorithms can provide a viable route for better robustness and stability, as well as future study of maximum entropy approaches.

### 2.3 Comparison and Summary

According to the research papers analyzed, it is clear that, while DQN is a cutting-edge model, it falls short in situations with high dimensions and continuous state space, so a newer and better method, DDPG, was proposed. However, due to the highly sensitive hyperparameters and brittleness of DDPG, SAC was proposed. For this study, we propose to build all three algorithms in the environment described by Zhao et al. (2020) and compare their performance.

## 3 Methodology

The research incorporates the development of an RL Agent with Deep Q-Networks trained in the Lunar Lander environment, as well as the deployment of DDPG and SAC in the same environment, therefore a large number of experiments were carried out, and the research was organised in a similar fashion. Despite the fact that the industry primarily employs one of the three major approaches, KDD, CRISP-DM, and SEMMA (Azevedo and Santos (2008)), we present a new methodology that is more appropriate for this research. Keeping this in mind, the research will be carried out in the following steps.

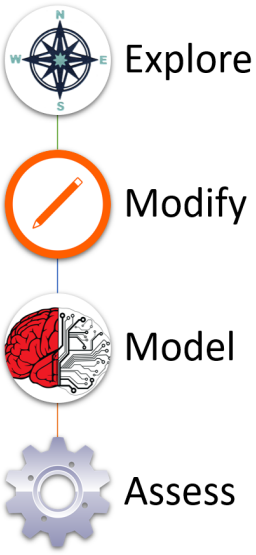


Figure 2: EMMA Structure for RL Agent Training

## 3.1 Explore

Extensive exploration was carried out to comprehend the game's number of actions, as well as the environment's input space. For neural networks, the number of output layers is based on this information.

The landing pad in Lunar Lander is always at the same location (0,0). About 100 to 140 points are awarded for going from the top of the screen to the landing pad at zero speed. If the lander goes away from the landing pad, it loses the reward it had previously received. If the lander crashes or comes to a halt, the episode is over, and the player receives an additional -100 or +100 points. +10 for each leg's touch with the ground. Every frame when the primary engine is firing costs you -0.3 points. It's worth 200 points to get it right. It is possible to land away from the landing pad. It's possible for an agent to learn to fly and land on its first attempt because the fuel supply is endless. Action is a two-dimensional vector of real numbers ranging from -1 to +1. -1 to 0 off, 0 to +1 throttle from 50% to 100% power are the first controls for the primary engine. The engine can't function at less than half its capacity. Fire the left engine from -1.0 to -0.5 and the right engine from +0.5 to +1.00[2].

[2]https://gym.openai.com/envs/LunarLanderContinuous-v2/

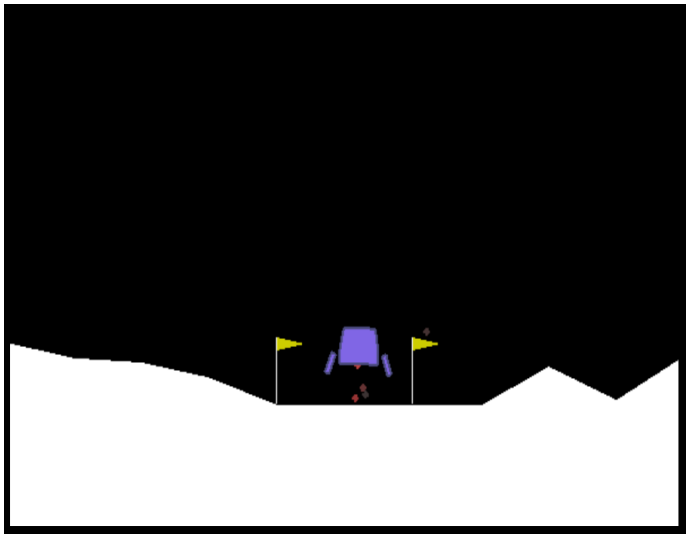

Figure 3: Lunar Lander Environment

## 3.2 Modify

The DummyVecEnv function from the stable baselines package was used to vectorize the environment prior to installing DDPG and SAC in a continuous state environment. Instead of training on one environment at a time, the RL agent can train on a stack of environments at once. The vectorized environment would not operate with continuous state environments since DQN from stable baselines library has a limitation of only working in discrete environments.
RL algorithms, like as DQN, DDPG, and SAC, do not require a training dataset; rather, every experience they encounter in the environment serves as their training dataset. For this study, the replay episodic memory for each algorithm was set at 50000 episodes, which can be lowered if we want the model to just use recent encounters. However, after this prior episodes will be erased and replaced by new encounters.

## 3.3 Model

The selected models DQN, DDPG, and SAC are deployed in the environment in the following step. We used the default hyperparameters for all three algorithms. In addition, we updated the batch size and the number of hidden layers for each of these techniques to improve the models' mean rewards. The stable baselines library's MlpPolicy, which has two layers of 64 nodes each, is used in all six models, although the tuned models have 256 nodes instead of 64.

## 3.4 Assess

In the absence of a training dataset, metrics like accuracy, precision, recall become obsolete to measure the model's performance in Reinforcement Learning. Hence, we have used mean reward and std. of rewards earned by the models over an interval of 100 episodes as major metrics to assess the models' performance. To select the best model amongst default and tuned models, various loss metrics like value loss, policy loss entropy are used.

# 4 Design Specification

In the Lunar Lander environment, the RL agents used DQN, DDPG, and SAC to auto identify the policies and land the rocket between the two flags. The algorithms are described in detail in the following section.

## 4.1 DQN

DQN estimates the Q-value function using a Neural Network. The input to a DQN is a raw image of the current state of the environment, which is processed by many convolution layers and fully connected dense layers. As output, Q-values for the agent's next best action are provided.
Additionally, two strategies are critical for DQN training:
1. Replay Experience: Because the training samples in a typical RL setup are highly correlated and less data-efficient, the network will have a harder time convergent. Adopting experience replay is one technique to overcome the sample distribution problem. In essence, sample transitions are saved and then randomly picked from the "transition pool" to update the knowledge.
2. Separate Target Network: The target Q Network is structured identically to the value estimation network. According to the pseudo code in figure 4 mentioned below, the target network is reset every C steps. As a result, the fluctuation is reduced, resulting in more stable training sessions.

**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t$ = 1,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**

Figure 4: Pseudo Code for DQN(Mnih et al. (2013))

## 4.2 DDPG

DDPG is a RL technique that combines Q-learning and policy gradients. As an actor-critic technique, DDPG utilizes two models: the actor and the critic. Instead of a probability distribution of actions, the actor is a policy network that takes the state as input and outputs the exact action (continuous). The critic is a network of Q-values that accepts state and action as input and outputs the Q-value. DDPG is a "non-standard" policy approach. DDPG is employed in the continuous action context, and the term "deterministic" refers to the actor computing the action directly, rather than using a probability distribution over actions.

**Algorithm 1** DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
**for** episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  **for** t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  **end for**
**end for**

Figure 5: Pseudo Code for DDPG(Lillicrap et al. (2015))

## 4.3 SAC

The Soft Actor Critic (SAC) algorithm optimizes a stochastic policy in an ad hoc fashion, bridging the gap between stochastic policy optimization and DDPG-style approaches. Entropy regularization is a critical element of SAC. The policy is trained to optimize the trade-off between expected return and entropy, which is a measure of the policy's randomness. This is closely related to the exploration-exploitation trade-off: increasing entropy results in increased exploration, which can speed up subsequent learning. Additionally, it can prevent the policy from convergent prematurely to a sub-optimal local optimum. Stable Baselines has employed an entropy coefficient (like in OpenAI Spinning or Facebook Horizon) in the code implementation, which is similar to the inverse of the reward scale in the original SAC work. The primary reason is that it prevents excessive mistake rates when changing the Q functions. To match the original paper, Stable baselines has used Relu activation function instead of default tanh activation function in MlpPolicy.

**Algorithm 1** Soft Actor-Critic

Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
**repeat**
  Observe state $s$ and select action $a \sim \pi_\theta(\cdot|s)$
  Execute $a$ in the environment
  Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
  Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
  If $s'$ is terminal, reset environment state.
  **if** it's time to update **then**
    **for** $j$ in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt $\theta$ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    **end for**
  **end if**
**until** convergence

Figure 6: Pseudo Code for SAC(Haarnoja et al. (2018))

# 5 Implementation

This section discusses how agents are implemented in the Lunar Lander environment. All three algorithms begin with two fully connected layers of 64 nodes each with a batch size of 32 by default and the tanh activation function, with the exception of SAC, which uses the Relu activation function. The tuned models have a batch size of 256, 50000 episodes of replay storage memory, and two layers of 256 nodes. The implementation is covered in detail in the following section.

## 5.1 Environmental Setup

The models were trained on an AMD Ryzen5900HS 3.3GHz processor coupled with 16GB DDR4 RAM. Python-3.7.11 was used to establish an anaconda environment via Jupyter Notebook. The models were developed using the Stable Baselines library version 2.10.2, which requires a base installation of tensorflow 1.15. The environment was produced using the version 0.21.0 of the OpenAI Gym library. Due to the fact that the Lunar Lander environment is a Box2D environment, installation of the Box2D library was also required. Tensorboard was used to evaluate the models' performance; it provides detailed information about model losses.

## 5.2 Implementation of DQN in Lunar Lander

As mentioned above the base architecture of the first model was 64*64 nodes with tanh activation function. DQN uses Adam optimizer to optimize the weights of the model. An object of class of 'Lunar Lander' environment was created and passed to the DQN model

object from Stable Baselines. Since DQN can operate on discrete environments hence we generated discrete object for environment. The internal architecture for a DQN model is shown below in figure 7.

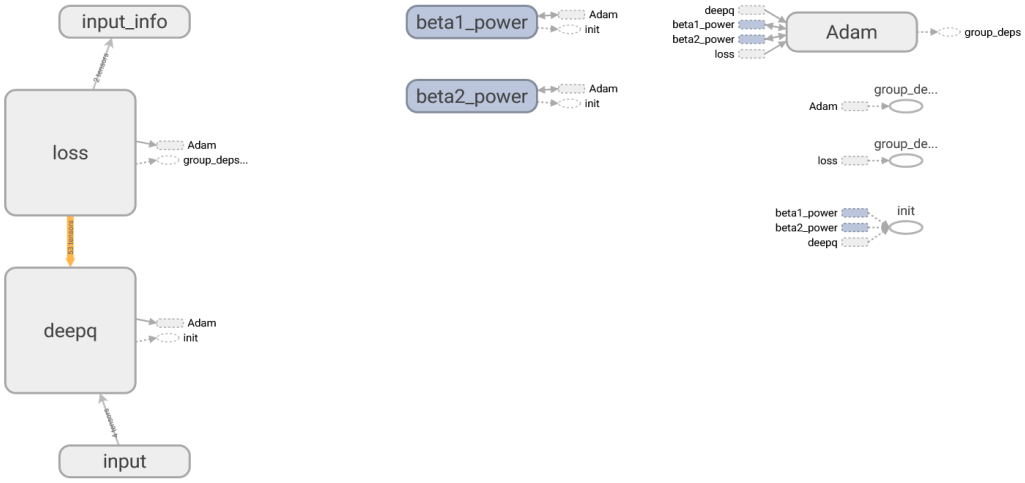


Figure 7: Architecture of a DQN Model

The input is the gym environment, raw pixels of the environment are directly fed to the model shown as deepq in 7, the difference between the predicted Q-values and a target Q-value from the current state of the environment creates loss, which is fed back to the algorithm for the next iteration, and the Adam optimizer updates the weights based on the information from loss.

## 5.3 Implementation of DDPG in Lunar Lander

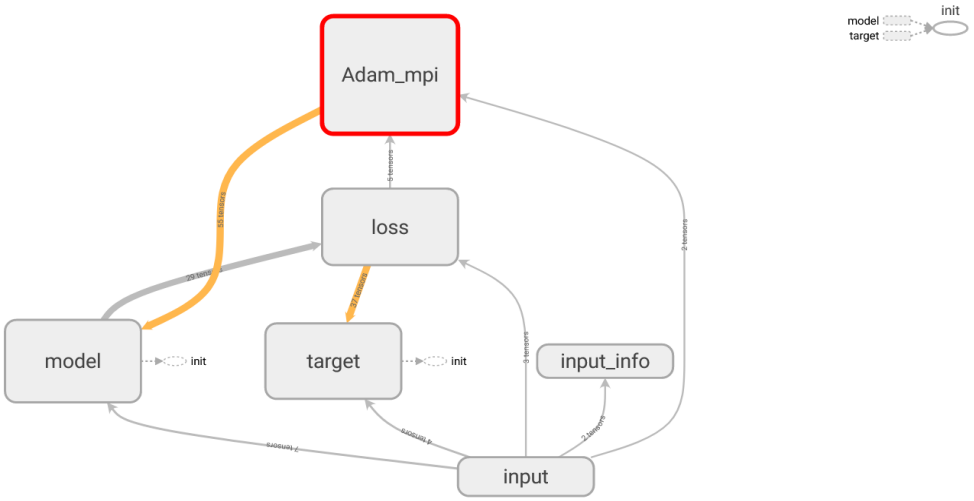


Figure 8: Architecture of a DDPG Model

Just like DQN, DDPG also had a similar architecture but DDPG requires extra parameters deal with noise generated from actions made by agent while interacting with the environment. OrnsteinUhlenbeckActionNoise function or the Ornstein-Uhlenbeck Process generates noise that is correlated with the previous noise, as to prevent the noise from canceling out or "freezing" the overall dynamics.
From the figure 8 we can observe that the critic and actor models are randomly initialized based on the environment's input state, and a target is also set. The model starts a random process and decides what to do based on current policy and noise. The agent obtains a reward and the current state of the environment as a result of its actions. The Adam optimizer attempts to minimize the losses perceived by the critic throughout the cycle and adjusts the actor's policy.

## 5.4 Implementation of SAC in Lunar Lander

We used two models of SAC with 64*64 fully connected nodes and the other one with 256*256 fully connected nodes with Relu activation function. To tune the performance of the model, we lowered the optimal learning rate from a default of 0.0003 to 0.0001 for the second model.

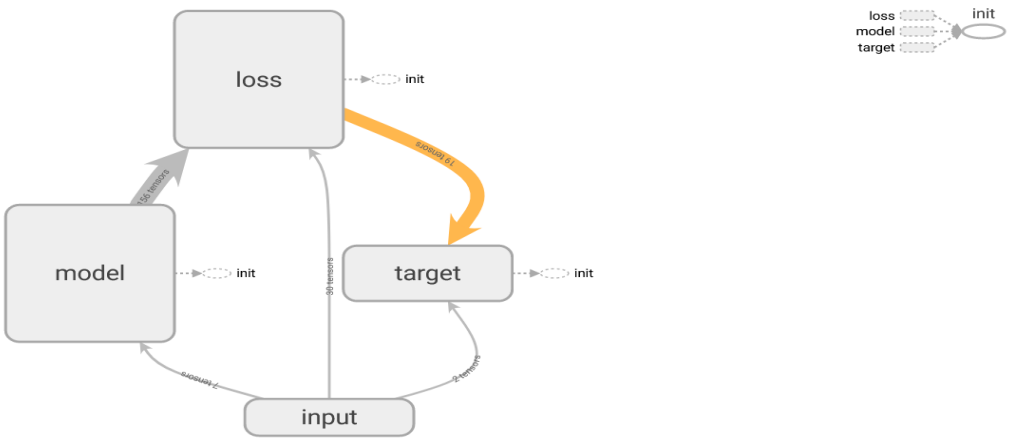


Figure 9: Architecture of a SAC Model

From the figure 9 above, we can observe that based on the initial state of the environment, the model initializes a policy and a Q-value function. Then the model takes an action after observing the state and collects the reward from the environment be it negative or positive. The action taken changes the state of the environment hence this is also taken as a feedback and the scenario is stored in the memory. Based on the state of environment, the model calculates the losses and updates the Q-function to calculate new target value.

# 6 Evaluation

A series of experiments were conducted to select the best performing models. The models were initially trained for 100,000 episodes. After implementing all six models in Lunar Lander environment, we evaluated the performance of the models using evaluate_policy which provides the mean of rewards earned by the model over a given number of episodes. We tested the model's performance for 10 episodes. After selecting the best model amongst DQN, DDPG and SAC, the models were again trained for 1 million episodes and best model was selected based on mean score earned over 100 episodes.

## 6.1 Default DQN vs Tuned DQN

After constructing models for DQN, we first evaluated the model's performance before training for 100,000 episodes. This was done to establish a baseline of scores for the models before to training so that we could measure the models' performance once training was completed.

```
mean_reward, std_reward = evaluate_policy(dqn_model_1, env, n_eval_episodes=10)
mean_reward_2, std_reward_2 = evaluate_policy(dqn_model_2, env, n_eval_episodes=10)

print(f'Mean reward: {mean_reward} +/- {std_reward:.2f}')
print(f'Mean reward: {mean_reward_2} +/- {std_reward_2:.2f}')
```

executed in 889ms, finished 14:50:43 2021-12-14

```
Mean reward: -456.3235224214692 +/- 114.15
Mean reward: -237.0796536180074 +/- 38.49
```

Figure 10: Initial Rewards for DQN before Training

After successful training over 100000 episodes, the first DQN model took 285.89s while the tuned model took 520.20s to complete its training. The models were again evaluated and it was observed that the tuned DQN model earned nearly 222 points in the environment clearly outperforming the model with default hyperparameters by a huge margin.

```
mean_reward, std_reward = evaluate_policy(dqn_model_1, env, n_eval_episodes=10)
mean_reward_2, std_reward_2 = evaluate_policy(dqn_model_2, env, n_eval_episodes=10)

print(f'Mean reward: {mean_reward} +/- {std_reward:.2f}')
print(f'Mean reward: {mean_reward_2} +/- {std_reward_2:.2f}')
```

executed in 16.5s, finished 15:04:26 2021-12-14

```
Mean reward: -4.184821919362525 +/- 21.47
Mean reward: 222.42588489069277 +/- 82.28
```

Figure 11: Final Rewards for DQN after Training

Figure 12 depicts the TD errors for the models during training. The TD error is how far the current prediction function deviates from this condition for the current input, and the agent works to reduce this error. As observed from the graph, the tuned model has diverged the least across the training cycles, but the trend from the graph for the first model shows a struggle to minimize error and has even departed from target more frequently.

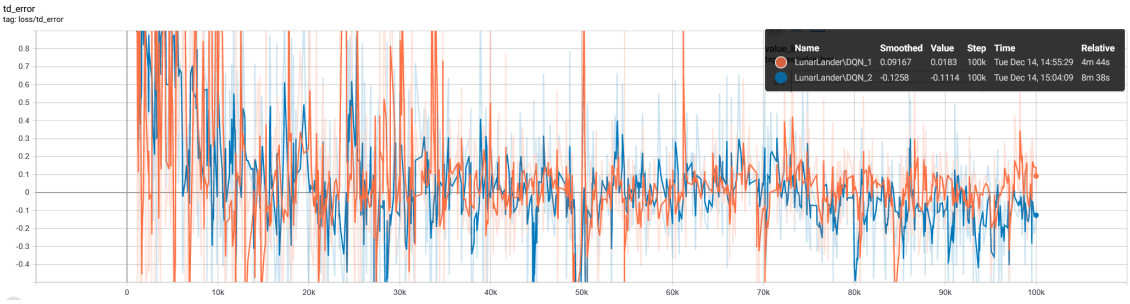


Figure 12: TD Error for Default DQN(orange) vs Tuned DQN Model(blue)

Figure 13 shows the improvement in rewards earned by the models during the training period.

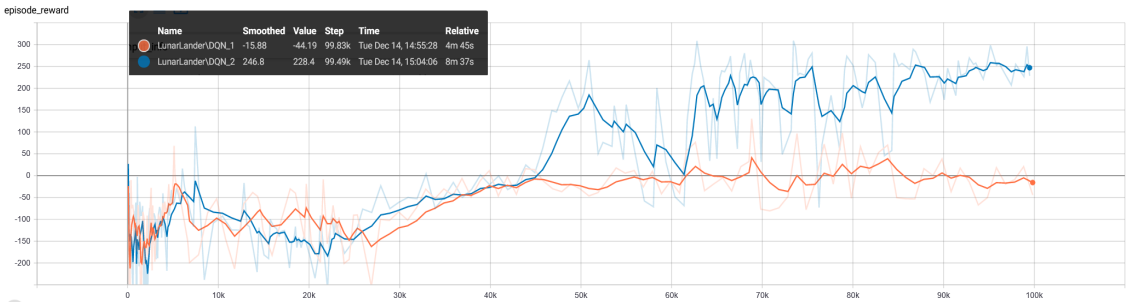


Figure 13: Rewards earned by models over training period

## 6.2 Default DDPG vs Tuned DDPG

After building models for DDPG, we tested the model's performance before training for 100,000 episodes. This was done to set a baseline of scores for the models before to training so that we could measure the models' performance once training was completed.

```
1 mean_reward, std_reward = evaluate_policy(ddpg_model_1, env_2, n_eval_episodes=10)
2 mean_reward_2, std_reward_2 = evaluate_policy(ddpg_model_2, env_2, n_eval_episodes=10)
3
4 print(f'Mean reward: {mean_reward} +/- {std_reward:.2f}')
5 print(f'Mean reward: {mean_reward_2} +/- {std_reward_2:.2f}')
```

executed in 2.22s, finished 15:04:29 2021-12-14

```
Mean reward: -271.21795654296875 +/- 101.47
Mean reward: -262.07061767578125 +/- 81.78
```

Figure 14: Initial Rewards for DDPG before Training

After successful training over 100000 episodes, the first DDPG model took 302.49s while the tuned model took 492.24s to complete its training. The models were again evaluated and it was observed that the models performed poorly and struggled to get a good score. This could be because of the reason that the models required more training .

```
mean_reward, std_reward = evaluate_policy(ddpg_model_1, env_2, n_eval_episodes=10)
mean_reward_2, std_reward_2 = evaluate_policy(ddpg_model_2, env_2, n_eval_episodes=10)

print(f'Mean reward: {mean_reward} +/- {std_reward:.2f}')
print(f'Mean reward: {mean_reward_2} +/- {std_reward_2:.2f}')
```

executed in 26.0s, finished 15:18:09 2021-12-14

```
Mean reward: -250.1200408935547 +/- 61.95
Mean reward: -160.09805297851562 +/- 78.48
```

Figure 15: Final Rewards for DDPG after Training

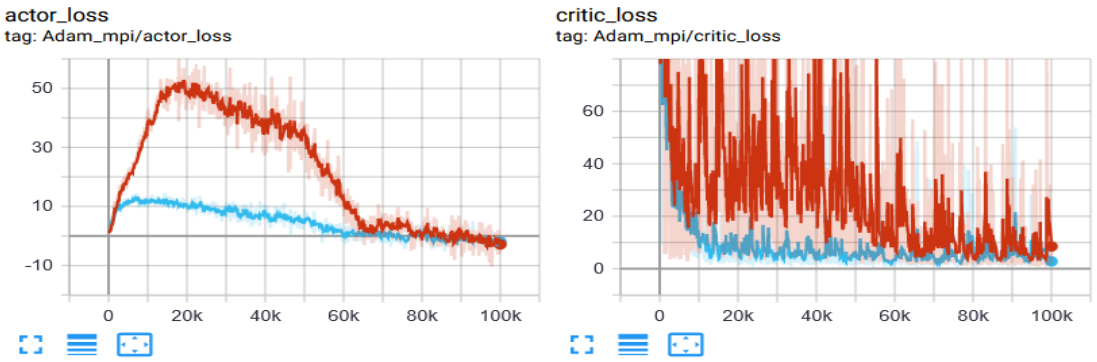


Figure 16: Actor and Critic losses for Default DDPG(red) vs Tuned DDPG Model(blue)

The losses generated by actor and critic models during training sessions are depicted in figure 16. The model's goal is to reduce these losses as much as feasible. The tuned model adjusts its weight quickly to reduce losses, making it a better model than the one with default hyperparameters.

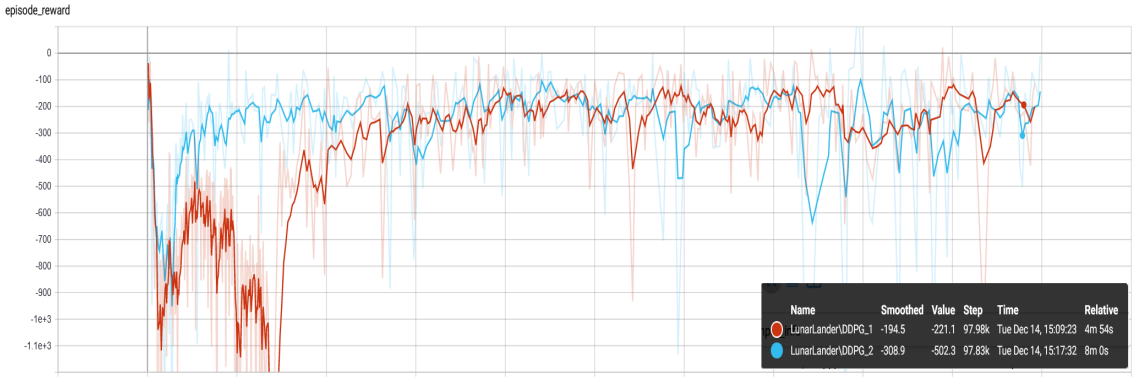


Figure 17: Rewards earned by DDPG models over training period

Figure 17 shows the improvement in rewards earned by the models during the training period but the models fail to achieve better scores than DQN models.

## 6.3 Default SAC vs Tuned SAC

To assess the performance of SAC models, we have used value loss function, policy loss function and entropy metrics.

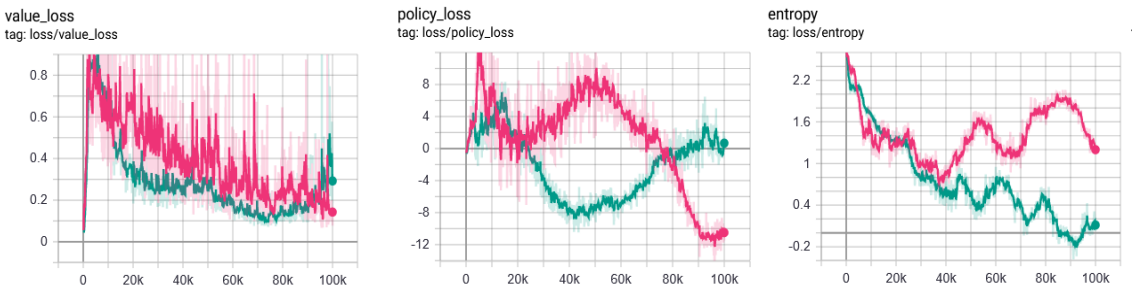


Figure 18: Loss Metrics for Default SAC(Pink) vs Tuned SAC(Green) models over training period

Entropy refers to the predictability of the agents action in the given environment. This is closely related to the policy's certainty about which action would provide the greatest cumulative reward in the long run: lower the entropy higher the chances of greater rewards.
Value Loss correlates to how well the model is able to predict the value for each state in the environment. For a good model, the value loss should converge when the rewards earned by the model stabilizes.
As value loss is about prediction of value for each state, Policy Loss correlates to how well the model is able to follow the policies of the environment in its pursuit of the goal.
From the information above we were able to conclude that the second model outperforms the first as the tuned model quickly converges its value losses, has lower entropy and has even lower policy losses in later stages of training.

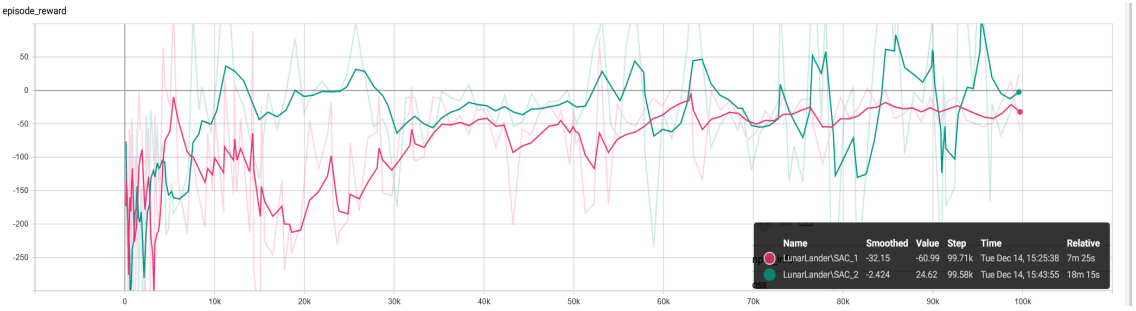


Figure 19: Rewards earned by SAC models over training period

The trends above conclude our analysis as the tuned model earns better rewards that the first model over training period.

## 6.4 Best of DQN vs DDPG vs SAC

After training each of the model for initial 100000 episodes, best performing model from DQN, DDPG and SAC was selected and trained again for a 1 million episodes. Their performance was then compared against each other.

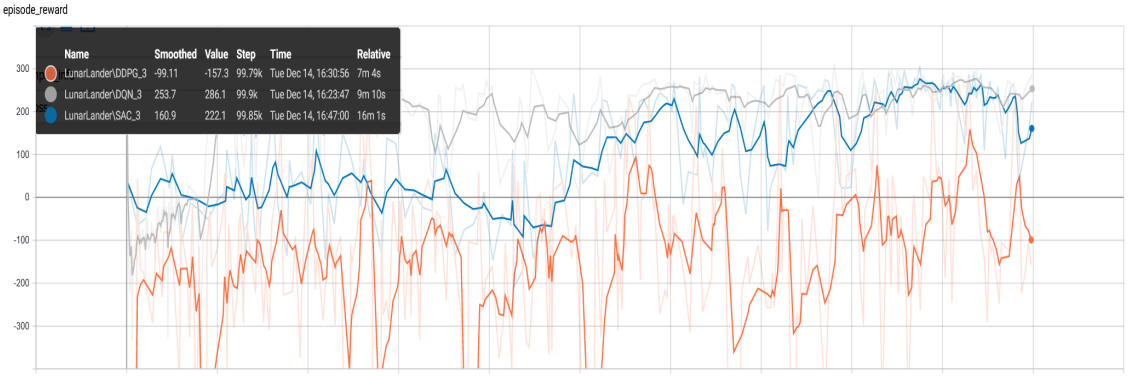


Figure 20: Rewards earned by each model over training period

The graph demonstrates that DQN initially outperforms DDPG and SAC, but SAC gradually catches up to DQN. The models were then assessed for 100 episodes using the evaluate policy function. The tuned SAC model outscored both DQN and DDPG by a large margin, gaining roughly 202 points across 100 episodes.

```
1 print(f'Mean reward for DQN: {mean_reward_dqn} +/- {std_reward_dqn:.2f}')
2 print(f'Mean reward for DDPG: {mean_reward_ddpg} +/- {std_reward_ddpg:.2f}')
3 print(f'Mean reward for SAC: {mean_reward_sac} +/- {std_reward_sac:.2f}')
```

executed in 15ms, finished 02:14:08 2021-12-15

```
Mean reward for DQN: -18.143590376806696 +/- 41.34
Mean reward for DDPG: -104.1529769897461 +/- 133.30
Mean reward for SAC: 202.7703094482422 +/- 106.53
```

Figure 21: Average Rewards Earned by models over 100 episodes

## 6.5 Discussion

According to the results of the preceding studies, SAC outperformed the other two algorithms tested. We chose DDPG and SAC for comparison against DQN because both algorithms are off-policy and employ Q-values for estimation, precisely like the DQN algorithm. We could have used on-policy algorithms such as Actor to Critic (A2C), Proximal Policy Optimization (PPO), and so on, but on-policy algorithms are better suited to places where the agent has more to explore about the environment, and since the agent has very limited exploration tasks to do in the Lunar Lander, on-policy algorithms were not used.

For this research, we had used Stable baselines library for the development of our agents but a drawback of using this library was that a very few documentation is provided on how to finely tune the hyperparameters of the models and even change the architecture of the hidden layers in the model. Instead we can use Tensorflow's reinforcement learning library which gives more freedom to the users to play around with the layers and activation functions of the models. Given the time and effort for the research using Stable baselines was a good choice as the models are quick to deploy and easy to save and load for future use.

# 7 Conclusion and Future Work

This research sought to solve the limitations of Deep Q-Networks by utilizing other off-policy algorithms, DDPG and SAC, and to provide a superior approach to Zhao et al. (2020) .'s research. To address this, we developed six models, two of each type of algorithm, and determined the best model for each type. We then compared the performance of the best DQN, DDPG, and SAC models. We successfully found that SAC algorithm had an upperhand over DQN model as it performed well during evaluation stage with a mean score of 202 while the best DQN and DDPG models secured -ve points over 100 episodes of evaluation environment. The fall in performance of DQN model could be because of the limitations mentioned in the paper above or training the model for longer durations could have yielded better results but in a span of 1 million training episodes, SAC outclassed DQN model. The two SAC models nearly scored same rewards after training of 100000 episodes but the value losses, policy loss and entropy helped in choosing a better model of the two. Thus proving that the tuned models performed much superior to the ones with default hyperparameters.

In future work, we intend to use the stable baselines 3 library to train DQN models in parallel to accelerate the training process, as reinforcement learning agents frequently require a large number of simulations in their environment to surpass human-level performance, and multiprocessing the training could significantly reduce training time. We even intend to combine stochastic variant Monte Carlo Tree search algorithm mentioned by Zhao et al. (2020) in their research with the SAC algorithm and compare the results with the original ones. This would help the agent train with limited information available about the environment which is close to real world gaming problems as in the scope of this research, the model had complete information about the state of the environment but with Monte Carlo Tree Search this information would be limited. This would be beneficial because in real life gaming environments agents don't have complete access to the environments.

# 8 Acknowledgement

I would like to thank my supervisor Dr. Mohammed Hasanuzzamanfor his devoted guidance and helping me with the chosen topic. I'd want to thank him for his regular assistance and supervision, which helped the project's execution go smoothly.

# References

Azevedo, A. and Santos, M. F. (2008). Kdd, semma and crisp-dm: a parallel overview, *ISCAP - Sistemas de Informação - Comunicações em eventos científicos* . **URL:** *http://hdl.handle.net/10400.22/136*

Bellemare, M. G., Naddaf, Y., Veness, J. and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents, *Journal of Artificial Intelligence Research* **47**: 253–279.

de Almeida, L. A. and Thielo, M. R. (2020). An intelligent agent playing generic action games based on deep reinforcement learning with memory restrictions, *2020 19th*

*Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pp. 29–37.

Gorman, B. and Humphrys, M. (2007). Imitative learning of combat behaviours in first-person computer games.

Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, *International conference on machine learning*, PMLR, pp. 1861–1870.

Hausknecht, M., Lehman, J., Miikkulainen, R. and Stone, P. (2014). A neuroevolution approach to general atari game playing, *IEEE Transactions on Computational Intelligence and AI in Games* **6**(4): 355–366.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. and Meger, D. (2018). Deep reinforcement learning that matters, *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castañeda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K. and Graepel, T. (2019). Human-level performance in 3d multiplayer games with population-based reinforcement learning, *Science* **364**(6443): 859–865.
**URL:** *https://science.sciencemag.org/content/364/6443/859*

Jäger, J., Helfenstein, F. and Scharf, F. (2021). *Bring Color to Deep Q-Networks: Limitations and Improvements of DQN Leading to Rainbow DQN*, Springer International Publishing, Cham, pp. 135–149.
**URL:** *https://doi.org/10.1007/978-3-030-41188-6*$_1$2

Levine, S. and Koltun, V. (2013). Guided policy search, *in* S. Dasgupta and D. McAllester (eds), *Proceedings of the 30th International Conference on Machine Learning*, Vol. 28 of *Proceedings of Machine Learning Research*, PMLR, Atlanta, Georgia, USA, pp. 1–9.
**URL:** *https://proceedings.mlr.press/v28/levine13.html*

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

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning, *International conference on machine learning*, PMLR, pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing atari with deep reinforcement learning, *arXiv preprint arXiv:1312.5602* .

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

Schulman, J., Chen, X. and Abbeel, P. (2017). Equivalence between policy gradients and soft q-learning, *arXiv preprint arXiv:1704.06440* .

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

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D. and Graepel, T. e. a. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play, *Science* **362**(6419): 1140–1144.

Sugiyama, M. (2015). *Statistical reinforcement learning: modern machine learning approaches*, CRC Press.

Tesauro, G. (1995). Temporal difference learning and td-gammon, *ICGA Journal* **18**(2): 88–88.

Van Hasselt, H., Guez, A. and Silver, D. (2016). Deep reinforcement learning with double q-learning, *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

Zhao, Y., Borovikov, I., de Mesentier Silva, F., Beirami, A., Rupert, J., Somers, C., Harder, J., Kolen, J., Pinto, J. and Pourabolghasem, R. e. a. (2020). Winning is not everything: Enhancing game development with intelligent agents, *IEEE Transactions on Games* **12**(2): 199–212.