B. TECH THESIS

# An Analysis of Deep Q-Networks and Applications of Generative Adversarial Networks in Reinforcement Learning

Adhokshaja V. Madhwaraj
Mentor: Dr. Prabuchandran K. J

160010032@iitdh.ac.in, prabukj@iitdh.ac.in
Indian Institute of Technology Dharwad

**KEYWORDS**
Deep Reinforcement Learning; Deep Q-Networks; Generative Adversarial Networks;

## 1. Introduction

Reinforcement Learning (RL) is a derivative of Machine Learning that is highly influenced by ideologies of Human Learning. Humans learn from their surroundings through multidimensional sensory inputs including visual, auditory, textural, olfactory stimuli among others, and corresponding rewards to learn the optimal behavior in a given situation . Optimal behavior in RL essentially implies maximized rewards. Reinforcement Learning incorporates these basic ideas to enable an agent to learn the optimal policy to follow in an environment. In this project we analyze and present the Deep Q-Learning algorithm along with its most important variants (Refer section 6).

Generative Adversarial Networks (GANs) have been termed as the *most interesting idea in the last 10 years in Machine Learning* (Refer subsection 10.1). In this project we present an attempt to generate the transition probabilities of a simple Markov Decision Process with only data from the environment as received by the agent, using GANs in section 13. With this find, we hope to extend this process to the more complex MDPs in the space of Reinforcement Learning, and ultimately be able to generate valid Experience Replay samples via the GAN, which will significantly reduce memory requirement of the learning process in Deep RL tasks.

## 2. Fundamentals of Reinforcement Learning

### *2.1. Agent and the Environment*

Learning takes place as a result of the interaction between an agent and the world. The perception that the agent receives is not only used for understanding or interpreting the information/data, but also for acting, and furthermore for improving the agent's ability to behave optimally in the future to achieve the goal.

Reinforcement Learning is different from Supervised Learning in that the agent does not receive direct instruction from the environment that the action taken is the correct one, instead receives a reward, 'reinforcement', which is essentially an evaluation of

the action taken by the agent. The rewards in most RL environments are not obtained immediately after an action is taken, hence the agent sees and understands a reward only after a prolonged set of steps through the environment. Thus, our main aim is to develop a policy (sequence of decision rules) in order to maximize it's long-term reward.

Let us consider an environment $E$, that is defined by all the possible states belonging to $S$. The agent can access the actions that are given by the set of legal actions $A = \{a_1, a_2, ..., a_n\}$. The state in which the agent is at time $t$ is given by $s_t$ . The agent takes an action $a_t$, to obtain reward $r_t$, to reach state $s_{t+1}$.

Each state and its dimensions depend on the particular environment. In the case of simpler environments like Mountain Car or Lunar Lander (refer subsection 3.1), the dimension of the state is small, 2 and 8, respectively. In the case of Atari Games (refer subsection 3.2), every state is defined by an entire gray-scale image of size $80 \times 80$. Though there is a general discrepancy in the state representation and size variation, the concept at the heart of it all remains the same, to obtain a representation of the environment at any given time-slice and obtain the best action to be taken to maximize the eventual reward to be obtained.

We consider a sequence of steps through the environment from the starting state until the agent reaches the final state. The sequence can be defined as a Markov Decision Process (MDP).

### *2.2. Markov Decision Process*

A Markov Decision Process can be defined as a tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where $S$ is a set of states, $A$ is a set of actions, $P_{sa}$ are the transition probabilities, $\gamma$ is the discount factor, and $R : S \times A \to \mathbb{R}$ is the reward function.

In an MDP, we start in some state $s_0$, and choose an action $a_0 \in A$. As a result of our choice of action, the next state is decided by stochastically based on the transition probability $P_{(s_0,a_0)}$. Thence, we pick the next action $a_1$, and so on, to get a sequence as follows,

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} ...$$

Upon following this sequence of steps, the total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + ...$$

or, when we write the rewards only based on the states, can be written as,

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$$

Our final goal in Reinforcement Learning is to maximize the expected value of the total future discounted reward, that is,

$$\mathbb{E}[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...]$$

### *2.3. Policy*

The policy is the function $\pi : S \to A$, mapping the states to actions. The value function can be defined based on a particular policy that the agent follows.

### *2.4. State Value Function*

The value function of a state $V(s)$, measures how good it is for an agent to be in state $s$. As we have seen, in most RL environments, rewards are not obtained in every state, or for every transition that the agent makes in the environment. Thus, the value of a given state depends on the states it *visits* in the future. The states in the future are deterministically/stochastically decided by the series of actions that the agent takes in the environment, which are taken based on a policy $\pi$.

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

Here the value function $v_\pi$ is calculated as the expected reward obtained over the upcoming states and actions once the agent is in the given state $s$. $v_\pi$ is called the state-value function of a policy $\pi$. The rewards are weighted by $\gamma$. This discount factor is included to account for the uncertainty of obtaining the reward in the future, and also the influence of the future states on the value of the current state. This scheme of evaluating future rewards is known as *Future Discounted Returns*.

### *2.5. State-Action Value Function*

The action-value function is defined over a state-action pair (s, a). This is the value of taking an action a, whilst in state s.

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

$q_\pi$ denotes the expected return that is obtained by starting at state $s$, and picking action $a$, and henceforth continuing to pick actions based on the policy $\pi$.

### *2.6. Policy Based methods*

Policy based methods revolve around the policy, consisting mainly of two steps - Policy Evaluation and Policy Improvement. The above two processes are carried out iteratively until convergence. Many algorithms search directly in the space of the policies, using gradient based methods called Policy-Gradient methods (ex. REINFORCE). These methods may converge slowly especially for noisy data, when trajectories are long and variance of returns is high.

### *2.7. Value Based methods*

Value based methods rely on the state-value and state-action value function to derive the optimal policy based on that. Value based methods mostly rely on TD-error based

formulations (ex. SARSA, Q-Learning).

## 3. Environments in Reinforcement Learning

Environments play the most important role in the testing of RL algorithms. In our analysis of RL algorithms and variants, we use OpenAI Gyms environments [1]. There are two sets of environments that we are analyzing our algorithms on - simple small-state environments (2-8 state dimensions), and Atari Games (RGB images, 210 $\times$ 160). Here, we give you a brief introduction to these environments to enable proper understanding of the results that are published in sections ahead.

### *3.1. Simple Environments*

#### *3.1.1. Mountain Car*

A car is on a one-dimensional track, positioned between two *mountains*. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Here, the reward is greater if you spend less energy to reach the goal.

- Number of states: 2 `(position, velocity)`
- Number of actions: 3 `(left, right, do nothing)`
- Reward Mechanism: Every time-step results in a -1 reward if the target is not reached, and reaching the target results in a 0 reward.

#### *3.1.2. Lunar Lander*

The main task in this environment is to land an outer-space rover safely on the ground within the demarcated landing pad. The Lander has a main engine that pushes it upwards, and two side engines, left and right, that propel it sideways. Landing outside the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

- Number of states: 8 `(position x and y, velocity x and y, angle and angular velocity, landing gear left and right)`
- Number of actions: 4 `(fire main engine, fire left engine, fire right engine, do nothing)`
- Reward Mechanism: Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If the lander moves away from the landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved 200 points.

#### *3.1.3. Cart Pole*

A pole is attached by an unactuated joint to a cart, which moves along a friction-less track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time-step that the pole remains upright. The episode ends

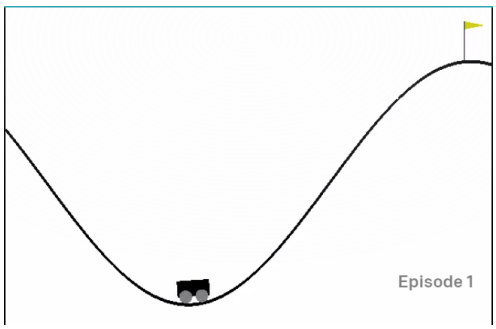
when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

- Number of states: 4 `(cart position, cart velocity, pole angle, pole velocity at the top)`
- Number of actions: 2 `(push cart left, push cart right)`
- Reward Mechanism: A reward of +1 is provided for every time-step that the pole remains upright, including the termination step.

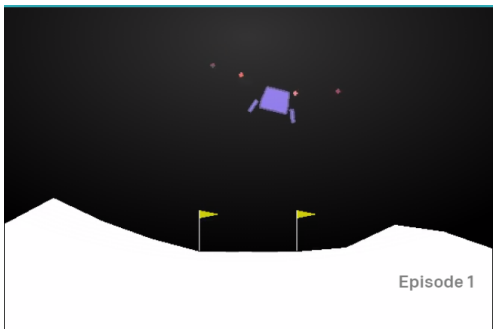
### *3.1.4. Acrobot*

The Acrobot system includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height.

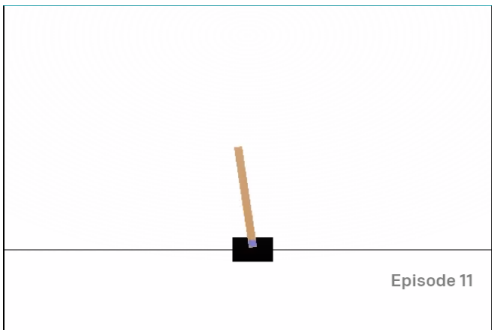
- Number of states: 6 `(sin and cos of the two rotational joint angles, and the joint angular velocities)`
- Number of actions: 3 `(Applying left torque, right torque, or no torque on the joint)`
- Reward Mechanism: A reward of -1 is provided for every time-step that the goal is not achieved



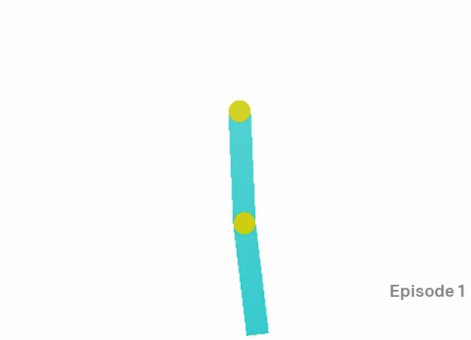

(a) Snippet of Mountain Car Environment




(b) Snippet of Lunar Lander Environment




(c) Snippet of Cart Pole Environment




(d) Snippet of Acrobot Environment

**Figure 1.** Snapshots of the Simple environments that were tested on during our experiments.

### *3.2. Atari Environments*

The Atari environment has been one of the most important benchmarks for research of Deep RL methods over the past few years. The Atari 2600 games are Arcade games that have actions ranging from 4-18 actions. The Atari 2600 framework built by OpenAI over the Arcade Learning Environment (ALE) has been widely used to test and benchmark algorithm performance. The Arcade Learning Environment (ALE) is a simple object-oriented framework that allows researchers and hobbyists to develop AI agents for Atari 2600 games. It is built on top of the Atari 2600 emulator *Stella* and separates the details of emulation from agent design.

Humans play games like Atari purely based on visual feedback. The main purpose to use frameworks like Atari is to enable Deep Learning Agents to understand and learn purely from visual stimuli like Images and video input.

In the Atari Framework, we directly work with raw Atari 2600 frames, which are $210 \times 160$ pixel images with a 128-color palette. This is highly demanding when it comes to computation. Each image typically has a portion of the image reserved for information like score, time remaining, lives remaining, etc. Thus for a learning agent, these statistics are not necessary for the training process. Thus it is important to pre-process the image in order to make it suitable for training.

#### *3.2.1. Pong*

The pong environment simulates a ping pong game between our agent and the computer. Every game of pong is until either player reaches 21 points (much like any game of ping pong.

- Number of states: `An image of 210 × 160`
- Number of actions: 6 `(Two actions (0,1) are dummy, actions 2 and 4 push the paddle up, actions 3 and 5 push the paddle down)`
- Reward Mechanism: A reward of +1 each time the agent wins a round, and -1 when the opponent (computer) wins a round. Max reward per episode is +21, while min reward is -21.

#### *3.2.2. Breakout*

The pong environment simulates a paddle that tries to hit a block of bricks, until no bricks are left. The agent has 5 lives (each life ends if the paddle can not reach the ball), before which the environment resets. The game ends when either the agent loses all its lives, or if there are no bricks left.

- Number of states: `An image of 210 × 160`
- Number of actions: 4 `(do nothing, fire ball, move left, move right)`
- Reward Mechanism: Each level of brick has a certain score. Hitting each brick adds the corresponding value to the reward.

#### *3.2.3. Space Invaders*

In the Space Invaders environment, the agent shoots down enemy space ships, until it is shot down itself. The agent can shoot, dodge and try to aim at the moving set of enemies.

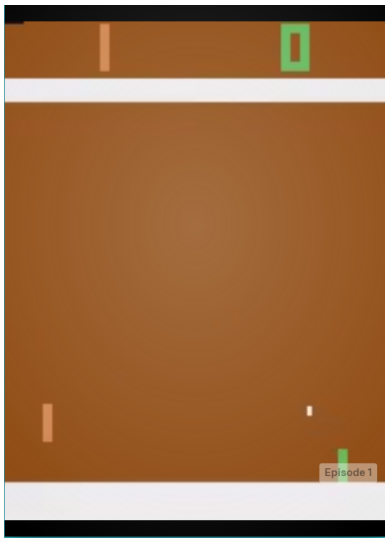
- Number of states: `An image of 210 × 160`

- Number of actions: 6 (do nothing, fire stationary, move right, move left, move right fire, move left fire)
- Reward Mechanism: The reward scheme is complex. A reward is obtained each time the agent destroys an enemy ship.

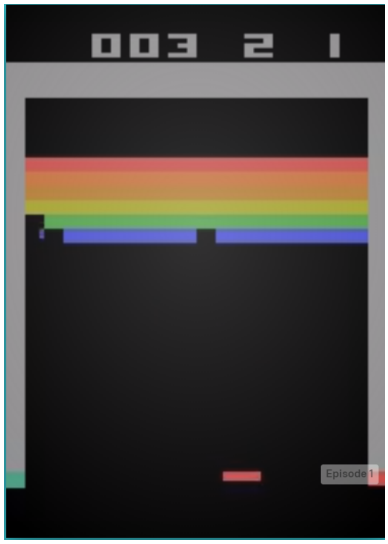
### *3.2.4. Enduro*

The Enduro environment is a racing game, with the objective to pass as many opponent vehicles as possible without getting passed yourself.

- Number of states: An image of $210 \times 160$
- Number of actions: 9 (do nothing, fire stationary, move right, move left, move right fire, move left fire)
- Reward Mechanism: A reward of +1 is obtained each time the agent passes a car, and -1 each time it gets passed. The total reward cannot reach 0.

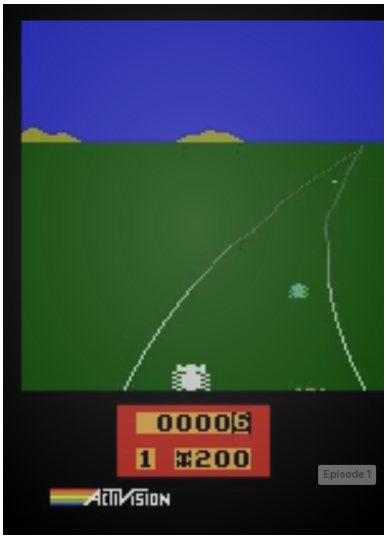


(a) Snippet of Pong Environment



(b) Snippet of Breakout Environment

(c) Snippet of Space Invaders Environment



(d) Snippet of Enduro Environment

**Figure 2.** Snapshots of the Atari environments that were tested on during our experiments.

## 4. Temporal Difference Methods

The Temporal Difference (TD) methods [2] belong to a class of model-free algorithms (those which do not consider the transition probabilities that are involved in transition from one state to another). TD methods use experience to solve the prediction problem. As we have observed earlier, reward mechanisms of many environments are such that rewards are skewed towards the later end of an episode. Thus, TD methods employ mechanisms to make calculated predictions about the future before the final outcome is known.

A simple TD update of the value function $v(s_t)$ subsection 2.4 is shown here. Similar updates can be made to the action-value function $q(s_t, a_t)$ subsection 2.5 as well.

$$v(s_t) \leftarrow v(s_t) + \alpha\Big[R_{t+1} + \gamma v(s_{t+1}) - v(s_t)\Big]$$

Here $v(s_t)$ is updated by the TD update rule, where $\alpha$ is the learning rate, $R_{t+1}$ is the reward obtained, and $v(s_{t+1})$ is the value function of the next state $s_{t+1}$. The term enclosed in brackets is called the TD error. The TD error at each time $t$ is the error in the estimate made at that time. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. The eventual goal of the TD algorithm is to reduce the TD error as much as possible, in other words, to estimate the values of the states, such that they match the actual value of the state.

Typical TD algorithms are online update algorithms, that is, the algorithm does not have to wait until an entire episode is completed. An episode is defined as the sequence of state, action pairs of the form $(s_0, a_1, s_1, a_2, s_2, a_3 ... s_{t-1}, a_t, s_t)$ where $s_0$ is the start state, and $s_t + 1$ is the terminal state. At each time-step $t$, the agent is in state $s_{t-1}$, and takes an action $a_{t-1}$, to move to state $s_t$. An episode is said to be completed if the agent moves into any of the designated terminal states (may be more than one).

## 5. Q-Learning Methods

One of the early breakthroughs in the Temporal Difference Methods was that of Q-Learning methods. The Q-Learning method is an off-policy, model-free reinforcement learning algorithm, which learns the optimal policy with the help of a greedy policy and behaves using the collective behavior of multiple agents.

To recap, the Q value which forms the basis of Q-Learning methods, is the state-action value function $Q(s, a)$, which is the measure of how rewarding taking a particular action $a$ is in a given state $s$. The learned state-action value directly approximates $q^*$ which is the optimal action-value function, independent of the policy being followed.

### *5.1. The algorithm*

At the heart of the Q-Learning algorithm is the Bellman Equation, which is a recursive expression that typically all MDPs satisfy. If we consider $q*$ to be the optimal q value, then the Bellman equation becomes the *Bellman Optimality Equation*. The Bellman Optimality Equation for q* is

$$q^*(s, a) = E_{s'}\Big[r + \gamma max_{a'} q^*(s', a')\Big]$$

The estimate can be replaced by a summation over $p(s', r|s, a)$, which is the transition probability of moving from state $s$, to $s'$, obtaining reward $r$. But since we are focusing on obtaining an off-policy estimate, we estimate this value directly from samples obtained from the emulator(environment), without explicitly estimating the transition dynamics.

The algorithm has the following parameters; $\gamma$ denoting the discount factor which accounts for the rewards obtained in the future. By multiplying future rewards by $\gamma$,

we are assigning a slightly lower value to rewards obtained in the future as compared to that obtained earlier, thus valuing what can be termed as a good start. The $\alpha$ parameter is the learning rate, typically chosen close to 0 to ensure stability of learning.

The Q values are randomly initialized to arbitrary values, typically 0. The algorithm steps through the environment, at each time interval $t$ in state $s_t$, the agent chooses an action at according to the $\epsilon$-greedy policy, and enters $s_{t+1}$ obtaining a reward $r_t$. Based on the Bellman update equation, $Q(s_t, a_t)$ is updated, by a simple value iteration step.

The Q-Learning method is a tabular method, as in, the values of the Q-function can be represented as a table, with states $s \in S$ as rows, and actions $a \in A$ as columns, and each table cell filled with $Q(s, a)$.

### *5.2. Q-Learning Pseudocode*

**Algorithm 1:** Q-Learning (off-policy TD)

Parameters: $\gamma$, $\alpha \in (0, 1]$, $\epsilon > 0$;
**for** *each episode* **do**
  initialize start state $s$;
  **while** *not done* **do**
    choose action $a$ using $\epsilon$-greedy policy;
    take action $a$, observe $r$, $s'$;
    $Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma max_{a'} Q(s', a') - Q(s, a)]$ $s \leftarrow s'$
  **end**
**end**

### *5.3. $\epsilon$-greedy policy*

The $\epsilon$-greedy policy can be defined as a semi greedy policy. $\epsilon$ is a small number belonging to (0, 1), closer to 0. Under the $\epsilon$-greedy policy of choosing actions, a random action is picked with a probability of $\epsilon$, while an action with respect to the Q-value is picked with a probability $1 - \epsilon$. This kind of scheme is chosen in order to balance the exploration vs exploitation dilemma. The $\epsilon$ value is chosen to be a high value closer to 1 initially to encourage exploration. $\epsilon$ is decayed slowly, eventually to a very small value near 0, to ensure that actions are taken according to the Q values.

### *5.4. The need for Deep Q-Networks*

As we have seen that Q-Learning methods are implemented in a tabular fashion, an astute reader will recognize the immediate drawback of this method as we move on to real-life environments where the state-space is much larger.

Traditional Q-Learning Methods are highly space-intensive, with a requirement of $(|S| \times |A|)$, and are not suitable with states with continuous values. Furthermore, this presents the challenge of exploring each and every state that belongs to the state-space. Though there are workarounds including Tile-coding among others, these methods are inefficient and do not solve the problem accurately.

Deep Q-Networks [3] help solve this problem by being able to learn directly from raw-inputs like sensory inputs. Most problems that can be solved through Q-Learning require labelled data, generally hand-crafted features that do not encompass the entire environment state. The deep network acts as a nonlinear function approximator of the

Q function of highly complex environments.

## 6. Deep Q-Networks

Deep Q-Networks (DQNs) are neural networks that learn to approximate the Q-function corresponding to an environment learning only from the states, and rewards obtained from the environment. The Deep Q-Network takes states as input and gives the Q-values corresponding to each action as output. Though DQNs were built to tackle large state-spaces and higher dimensional states, as an initial task, we implemented DQN on simpler tasks, to understand the exact working of the algorithm. Before we look at the results and inferences, it is important to understand some features and drawbacks that DQNs solved.

### *6.1. Challenges faced by RL coupled with DL*

Though it has been shown that off policy Q-function approximation using neural networks diverges, the most recent research and innovation has been focused on mitigating this divergence.

The primary challenges are listed below:

(1) Deep Learning solutions expect independent and identically distributed (iid) samples as input to prevent highly correlated data. In RL environments, data that is generated is highly correlated as consequent inputs are generated by consecutive states, thus a strategy is required to maintain iid nature, and ensure that each batch maintains the same/similar distribution.

(2) In Q-Learning, the target is constantly changing, thus presenting a moving target. This results in instability of the network, as both the inputs and outputs are constantly changing. As seen in the Bellman Optimality rule, the value of $Q$ depends on the value of $Q$ itself, thus chasing a non-stationary target.

### *6.2. Solutions to challenges*

The paper on DQN [3], presented two solutions for the two challenges presented above. The solutions are the introduction of an Experience Replay buffer and a Target Network, respectively.

#### *6.2.1. Experience Replay*

Most of Human learning happens through recollection and learning from past experience. Since we are trying to mitigate the problem of correlation, we randomly sample a set of experiences from an Experience Replay Buffer.

Each entry of the Experience Replay Buffer consists of the following $(s_t, a_t, r_t, s_{t+1}, done)$. The *done* variable suggests whether the episode is completed. The experience replay buffer is implemented as a queue, with a FIFO (first in, first out) policy.

The random sampling of the replay entries ensures a stable batch reducing the factor of correlation, and improves the iid property of the data. Also, there is a high possibility of the same sample being used multiple times during training, which results in higher data efficiency. As the experience replay buffer is randomly sampled from,

the behavior distribution of the network is averaged, and thus prevents the network from getting stuck at local minima, or even diverging drastically.

#### *6.2.2. Target Network*

The paper suggests another solution to the problem of the moving target by introducing another network, the target network. Let us assume that the parameters of the training neural network (trained in an online fashion, with each environment step) as $\theta$, and the parameters of the target network as $\theta^-$. The training network is constantly updated, while the other network follows this, albeit with a lag of $c$ steps, during which its weights are frozen. The target network is used to obtain the $Q$ values. Since the target network is fixed for $c$ steps, it presents a stationary target and stymies the challenge of instability.

Further challenges faced were that of varying rewards in each environment. This was handled by a clipping scheme. Any reward above 1, or below -1, was clipped to 1 and -1 respectively.

The DQN algorithm that incorporates all these changes is shown below. The Atari framework requires preprocessing of the states (refer subsection 9.1).

### *6.3. Algorithm Pseudocode*

**Algorithm 2:** Deep Q-Learning with Experience Replay

Initialization;
$\gamma$, $\alpha \in (0, 1]$, $\epsilon > 0$;
Initialize Experience Replay Buffer $M$ with capacity $N$;
Initialize training and target networks, $Q$ and $\hat{Q}$, with weights $\theta$ and $\theta^-$, $\theta = \theta^-$;
**for** *each episode* **do**
  Initialise start state/image $x$, and preprocess to obtain $s$;
  **while** *not done* **do**
    Choose action $a$ using $\epsilon$-greedy policy;
    Execute action $a$, observe $r$, and new state/image $x'$;
    Preprocess $x'$ to obtain $s'$;
    Store transition $(s, a, r, s', done)$ in $M$;
    Sample random mini-batch of transitions $(s_i, a_i, r_i, s_{i+1}, done_i)$;

$$y_i = \begin{cases} r_i, & \text{if } done_i \\ r_i + \gamma max_{a'} \hat{Q}(s_{i+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

    Perform Gradient Descent step on $\left(y_i - Q(s_i, a_i; \theta)\right)^2$ wrt network weights $\theta$;
    Every $c$ steps, reset $Q = \hat{Q}$
  **end**
**end**

## 7. Improvements to Deep Q-Networks

### *7.1. Double DQN*

To understand Double DQN, we should initially explore the idea of Double Q-Learning first. The idea of Double Q-Learning was first proposed by van Hasselt [4] in 2010.

#### *7.1.1. Double Q-Learning*

The *max* operator in standard DQN uses a common network for selecting the action and evaluating the corresponding value of the action. This generally leads to the overestimation of values, resulting in over-optimistic value estimates. To prevent this, we decouple these tasks of selection, and evaluation.

#### *7.1.2. The Algorithm*

The algorithm uses two Q-functions, $Q_1$ and $Q_2$ for the tasks of evaluation and selection of the action. For each update one of the networks is used for the greedy selection of the action, and the other is used for the evaluation of the Q-value based on this selection. A generic update rule considering the update of value of $Q_1$ is,

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, argmax_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)]$$

The original Double Q-Learning algorithm employs a stochastic process at each step through the environment to choose which network is to be used for which task (selection or evaluation).

#### *7.1.3. Double DQN Algorithm*

The idea of Double DQN [5] is to reduce the overestimations by decomposing the *max* operation in the target into action selection and action evaluation. Although the networks are not completely disjoint, the target network in the original DQN network can be used as the second network. The update rule for Double DQN can be defined as follows:

$$y_i^{DoubleDQN} = \begin{cases} r_i, & \text{if terminal step} \\ r_i + \gamma \hat{Q}(s_{i+1}, argmax_a Q(s_{t+1}, a, \theta_i); \theta_i^-) & \text{otherwise} \end{cases}$$

The rest of the algorithm is exactly as that of DQN, presented in algorithm 2. The target network continues to be a periodic copy of the training network.

### *7.2. Dueling DQN*

The Dueling DQN architecture [6] brought about a more drastic change to the DQN scene, as it introduced a new architecture, one termed as the Dueling architecture. The Dueling architecture explicitly splits the representation of state values, and the state-dependant state-action values. The Dueling network consists of two separate streams, called the *value* and *advantage* streams, while sharing a common convolutional feature learning module.

The Dueling architecture was developed with the intuition that in several RL environments, there are multiple states such that taking any action does not have any particular effect on the environment, or the reward. In the case of the Lunar Lander, the most important states are those that are close to the landing pad, and any state that is towards the top of the box environment, is not very relevant from a reward perspective.

#### *7.2.1. Architecture and Algorithm*

The algorithm utilises the definitions of Value and Q-functions. We define a new quantity called the Advantage function, which relates the value and Q functions as

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

$$Q^{\pi}(s,a) = A^{\pi}(s,a) + V^{\pi}(s)$$

We know that $\mathbb{E}_{a=\pi(s)}[Q^{\pi}(s,a)] = V^{\pi}(s)$. We shall use this knowledge to design the network and the streams of value and advantage layers. We may be tempted to directly design the network such that,

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

Note that this expression requires us to replicate the value function across all actions, since the value function remains constant across all actions. Since all these values are estimated, we can not be sure that the values $Q(s,a;\theta,\alpha,\beta)$, $V(s;\theta,\beta)$, and $A(s,a;\theta,\alpha)$ provide reasonable estimates of the actual values of the same.

A disadvantage of the above representation of $Q$ is that $Q$ is not identifiable. This means that given a value of $Q$, we cannot recover the individual values of $V$ and $A$. To mitigate this we add a value to the value stream and subtract the same value from the advantage stream. We use an alternative representation of the same equation as,

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left( A(s,a;\theta,\alpha) - \frac{1}{|A|} \sum_{a'} A(s,a';\theta,\alpha) \right)$$

It is important to note that these changes are incorporated and implemented as part of the network itself, and not as a separate computational implementation. This representation is shown to increase the stability of the model. This model also presents the unique premise of being able to make decisions based purely on the advantage function, since the value function is a constant for a given state across actions.

### *7.3. Prioritised Experience Replay*

Prioritised Experience Replay [7] was proposed as a unique sampling strategy of the Experience Replay (refer subsubsection 6.2.1) to ensure that samples that have more impact on the learning are used more often than others that are less relevant. The key

idea was to increase the replay probability of experience tuples that have a high expected learning progress (as measured via the proxy of absolute TD-error $\delta$) section 4. This led to both faster learning and to better final policy quality across most games of the Atari benchmark suite, as compared to uniform experience replay.

#### *7.3.1. Algorithm and Implementation*

The most important aspect of the priority of a sample is the criterion by which it is measured. An ideal criterion would be to measure the amount by which the agent learns from a given transition. As we obviously have no way of directly measuring this, we use a suitable proxy in the form of the TD-error $\delta$, which indicates how surprising or unexpected an action is.

An issue with greedy sampling based on TD-errors is that a sample with low TD-error, has a very low probability of being replayed multiple times. This method is particularly susceptible to noisy spikes in rewards. Greedy prioritization focuses on a small subset of the experience: errors shrink slowly, especially when using function approximation, meaning that the initially high error transitions get replayed frequently. This lack of diversity makes the system prone to over-fitting.

This is mitigated by using a stochastic sampling method that interpolates between purely greedy sampling, and purely random sampling. Each transition $(s_i, a_i, r_i, s_{i+1}, done_i)$ is assigned a priority value, $\delta$ (which is the TD-error of the transition). To avoid cases where $\delta = 0$, we add a small constant $e$, such that $p_i = \delta + e$. The probability of sampling a transition i, can be written as,

$$P(i) = \frac{p_i^\alpha}{\sum_k p_i^\alpha}$$

The value $p_i$ is the priority of transition, while the parameter $\alpha$ decides the extent of greediness. At $\alpha = 0$, the scheme becomes purely random, while at $\alpha = 1$, the scheme is purely greedy. A value of $\alpha = 0.6$ is suitable in experiments, to ensure a partly greedy scheme.

To minimize memory usage, a *sum-tree* data structure is used (where each node is the sum of its children, with the leaf nodes having the values of the priorities). This data structure implementation ensures efficient updation and sampling.

## 8. Analysis of Deep Q-Networks and its variants on Simple Tasks

The DQN algorithm for simple tasks remains the same as presented in the above pseudocode, minus the preprocessing. In the case of simple tasks, we use the states obtained from the emulator as it is, without performing any preprocessing.

### *8.1. Architecture for simple tasks*

The DQN architecture for simple tasks involves two neural networks, the training and target networks, both having the same architecture. In our experiments we used a two-hidden layer architecture for all the simple tasks, while slightly altering the hyperparameters, as well as the number of units in each of the Fully-Connected Dense layers. Each layer was appended with a ReLU activation layer, with `l2` kernel regular-

ization (to prevent over-fitting and increase the generalization ability of the network). All the networks for the simple task were built using `Keras` module

The input layer had inputs as the entire state as obtained from the environment (without any transformation/feature extraction). The output layer has units equal to the number of actions, with a Linear activation, since we need the exact Q-values as estimated by the network.

The Dueling architecture requires the splitting of the layers after the dense layers. We used a single unit Dense layer for the Value stream and then used `expand_dims` to obtain the same value across all actions. For the advantage stream, we used a Dense Layer having units equal to the number of actions. These two layers' outputs were added using `Lambda` layers.

### *8.2. Loss Function*

According to the paper by *Mnih, et al.* [3], the neural network behaves like a function approximator for the Q-values. The Q-network can be trained by adjusting the weights *theta* of the network at each iteration by trying to reduce the mean-squared error in the Bellman Equation, where the optimal target values $r + \gamma max_{a'} Q(s', a'; \theta)$ are substituted by approximate target values $y = r + \gamma max_{a'} Q(s', a'; \theta^-)$, using the parameters from *some* previous iteration.

Thus, the loss function for the network at a given time iteration $i$, can be shown by,

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[ (y - Q(s, a; \theta_i))^2 \right]$$

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[ (r + \gamma max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i))^2 \right]$$

This error function is highly similar to the TD-update rule, defined for the Q-Learning method. The Q-Learning update rule can be obtained if we update the weights of the network after each iteration, and set $\theta_i^- = \theta_{i-1}$

### *8.3. Optimization Strategies*

The networks use either `RMSProp` or `Adam` optimizers. In our experiments, both these optimizers gave highly similar results, leading us to conclude that this did not contribute much to the algorithms success.

In our experiments, we used a simple $\epsilon$-annealing policy as described below

$$\epsilon = \begin{cases} \epsilon - \epsilon_{decay}, & \text{if } \epsilon > \epsilon_{min} \\ \epsilon_{min} & \text{otherwise} \end{cases}$$

$$\text{at t=0}, \epsilon = 1.0$$

$$\epsilon_{min} = 0.01, \epsilon_{decay} = 0.001$$

The training and target networks operate with a lag as mentioned above. The network weights are made equal after a constant $c$ steps through the environment. There is an alternative partial-copy strategy where we have a small constant $\tau$. We update the target based on the following equation:

$$weights_{target} = \tau * weights_{target} + (1 - \tau) * weights_{training}, \tau = 0.01$$

The primary importance of this equation is to retain some important weights that the target network might have attained.

### *8.4. Training*

The models during training were saved with the weights frozen. The models were saved at the start (initial), mid-way, and final. The best model after 75% of the iterations was selected in a greedy fashion based on the total reward of the episode. These models were later used to generate statistics where the frozen model weights were used to generate the episodes, and the scores were recorded, and tabulated. Also training reward sequences and iterations taken to complete the task were recorded, and graphed. We present these results in a section below. Code snippets were referenced from [8]. The models took the following amounts of time on average for each of the algorithm variants:

- Mountain Car : $\sim$ 25 minutes for 500 episodes (one task)
- Lunar Lander : $\sim$ 50-60 minutes for 500 episodes (one task)
- Cart Pole : $\sim$ 15 minutes for 500 episodes (one task)
- Acrobot : $\sim$ 30 minutes for 500 episodes (one task)

The hyperparameters are presented in Table 1.

| Hyperparameter | Explanation | Mountain Car | Lunar Lander | Cart Pole | Acrobot |
|---|---|---|---|---|---|
| Batch_size | the number of samples collected from the experience replay buffer for training | 64 | 64 | 64 | 64 |
| replay_memory_size | the total number of experience replay samples to be stored | 100000 | 1000000 | 100000 | 100000 |
| Target_update_after | the number of steps after which the target network is equated to the training network | 400 | 200 | 100 | 200 |
| Gamma | discount factor | 0.99 | 0.99 | 0.99 | 0.99 |
| Dense_layer_1 | the number of units in the first fully-connected layer of the DQN | 32 | 256 | 32 | 128 |
| Dense_layer_2 | the number of units in the first fully-connected layer of the DQN | 32 | 256 | 32 | 128 |
| Learning_rate | learning rate of the RMSProp/Adam optimizer | 0.0005 | 0.0005 | 0.001 | 0.001 |
| L2_regularizer | the regularizer used | 0.01 | 0.01 | 0.01 | 0.01 |
| epsilon_initial | initial value of the epsilon value | 0.5 | 1 | 1 | 1 |
| Epsilon_decay | the constant with which epsilon is reduced with each step | 0.001 | 0.001 | 0.001 | 0.001 |
| Epsilon_final | the minimum value of epsilon at which it is maintained | 0.01 | 0.01 | 0.01 | 0.01 |
| Replay_start_after | the minimum number of steps before learning, this time is used to fill the buffer | 64 | 64 | 64 | 64 |
| Number_of_episodes | total number of complete episodes that the algorithm is left to run | 500 | 500 | 500 | 500 |
| Number_iterations | total number of steps of a given environment | 200 | 1000 | 500 | 500 |

**Table 1.** List of Hyperparameters involved in the training of the networks

### *8.5. Results of Simple Tasks*

#### *8.5.1. Explanation of the tables presented below*

Each of these models presented below were frozen during the training period.

- The initial model is the neural network as soon as it was initialized. Naturally the actions that are picked are at random, hence these values represent a random policy.
- The midway policy, as the name suggests, is the model frozen at the midpoint of the training process. The model has learned the basics of the policy, but has to undergo refinement. Thus we can see a high standard deviation.
- The final policy is the model frozen at the end of the training process. This model yields the best results, as it has the maximum experience, and has minimum standard deviation.
- The best model was saved based on a greedy strategy of the total reward of an episode. The best model was saved for the best reward after the 75% mark. This was done to ensure that models that had one-off anomalies (spikes) in the total reward due to randomness would not be selected.

The models that were saved were loaded, and each environment was run for 200 episodes each. The rewards were collected, and the metrics were calculated. Two action-choosing policies were selected:

(1) Absolute Policy: The actions were chosen greedily purely based on the Q-values generated by the models, given the state.
(2) $\epsilon$-greedy Policy: The actions were selected based on an $\epsilon$-greedy strategy.

The results obtained are presented below.

*8.5.2. Mountain Car*

| Algo | Model | $\mu$-Absolute | $\sigma$-Absolute | $\mu$-$\epsilon$-greedy | $\sigma$-$\epsilon$-greedy |
|---|---|---|---|---|---|
| dqn | initial | -200.0 | 0.0 | -200.0 | 0.0 |
| | final | -156.375 | 34.651 | -153.445 | 37.0 |
| | best | -187.355 | 26.564 | -185.96 | 27.271 |
| double-dqn | initial | -200.0 | 0.0 | -200.0 | 0.0 |
| | final | -187.515 | 34.641 | -193.19 | 25.753 |
| | best | -126.32 | 38.507 | -130.43 | 38.044 |
| Dueling-dqn | initial | -200.0 | 0.0 | -200.0 | 0.0 |
| | final | -128.19 | 32.072 | -130.245 | 32.34 |
| | best | -161.045 | 36.481 | -162.565 | 35.962 |

*8.5.3. Lunar Lander*

| Algo | Model | $\mu$-Absolute | $\sigma$-Absolute | $\mu$-$\epsilon$-greedy | $\sigma$-$\epsilon$-greedy |
|---|---|---|---|---|---|
| dqn | initial | -162.482 | 36.233 | -158.887 | 45.988 |
| | final | 264.237 | 40.226 | 267.897 | 30.14 |
| | best | -51.298 | 172.811 | -32.329 | 181.31 |
| double-dqn | initial | -577.074 | 168.104 | -573.091 | 163.269 |
| | final | 249.505 | 51.21 | 257.166 | 31.338 |
| | best | 267.475 | 22.958 | 266.129 | 23.009 |
| Dueling-dqn | initial | -865.927 | 557.288 | -195.776 | 108.549 |
| | final | 232.966 | 87.964 | 241.798 | 75.058 |
| | best | 239.86 | 62.36 | 227.823 | 65.275 |

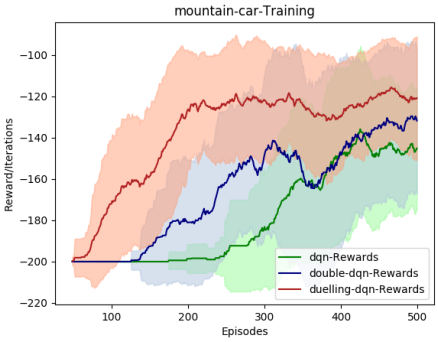
*8.5.4. Cart Pole*

| Algo | Model | $\mu$-Absolute | $\sigma$-Absolute | $\mu$-$\epsilon$-greedy | $\sigma$-$\epsilon$-greedy |
|---|---|---|---|---|---|
| dqn | initial | 10.29 | 2.756 | 10.445 | 2.887 |
| | final | 500.0 | 0.0 | 500.0 | 0.0 |
| | best | 500.0 | 0.0 | 500.0 | 0.0 |
| double-dqn | initial | 9.7 | 1.96 | 9.78 | 1.795 |
| | final | 500.0 | 0.0 | 500.0 | 0.0 |
| | best | 500.0 | 0.0 | 500.0 | 0.0 |
| Dueling-dqn | initial | 9.5 | 0.762 | 23.115 | 11.339 |
| | final | 462.97 | 80.979 | 472.0 | 72.861 |
| | best | 500.0 | 0.0 | 500.0 | 0.0 |

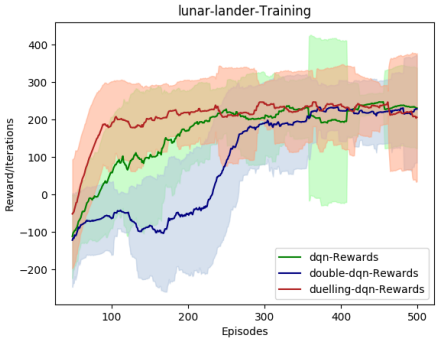
*8.5.5. Acrobot*

| Algo | Model | $\mu$-Absolute | $\sigma$-Absolute | $\mu$-$\epsilon$-greedy | $\sigma$-$\epsilon$-greedy |
|---|---|---|---|---|---|
| dqn | initial | -94.275 | 62.759 | -92.52 | 41.861 |
| | final | -85.74 | 20.574 | -83.365 | 19.055 |
| | best | -87.105 | 33.714 | -88.96 | 27.481 |
| double-dqn | initial | -499.695 | 2.669 | -498.35 | 11.227 |
| | final | -82.585 | 16.323 | -82.855 | 14.88 |
| | best | -75.74 | 11.979 | -79.675 | 16.401 |
| Dueling-dqn | initial | -500.0 | 0.0 | -455.56 | 70.344 |
| | final | -83.07 | 15.807 | -85.86 | 22.765 |
| | best | -84.475 | 16.827 | -83.665 | 15.378 |

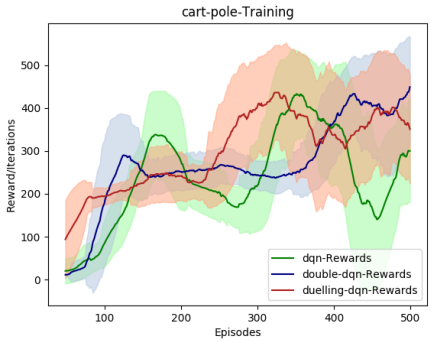
## *8.6. Observations and Inferences*



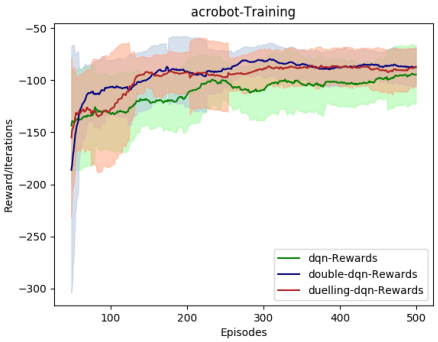

(a) Training Rewards of Mountain Car




(b) Training Rewards of Lunar Lander




(c) Training Rewards of Cart Pole




(d) Training Rewards of Acrobot

**Figure 3.** Rolling Mean with window size of 20 of rewards of each environment over the training period of 500 episodes each. The line represents the rewards, while the shaded region signifies the standard deviation of the window considered.

Through these experiments and results obtained, we have shown that all these environments can be solved fully using Deep Q-Learning methods, and its variants. One

thing that is imperative to consider is that these methods were not built to solve these smaller environments, rather more complex ones like the Atari emulators. Thus, if we observe the graphs plotted, we can observe the minor dips in the reward curves. This is probably due to over-fitting, since the use of neural networks on a simple task may be overkill.

Also we can observe that in general, Dueling DQN performs better than the other two variants. It can be seen in testing that the Dueling architecture is able to generalize fully the actions it needs to take, thus minimises the standard deviation. Also, there is a clear improvement in the performance of the models from the Vanilla-DQN to Double-DQN to consequently, Dueling-DQN.

## 9. Analysis of Deep Q-Networks and its variants on Atari Games

The DQN algorithm on Atari Environments involves preprocessing of the frames that the emulator provides as states/observations. In our experiments, we are using environments with NoFrameSkip, as the name suggests, not skipping any frames.

### *9.1. Preprocessing*

Preprocessing can be done in two alternatives. One, process the images after the images are obtained from the normal emulator, or two, to embed the preprocessing methodology into the emulator itself as a wrapper. The latter method is better as it simplifies the process of coding the actual algorithmic logic for the problem. This also makes the code more modular and thus reusable for implementation of the variants.

This process of preprocessing can be split into these sub-processes:

#### *9.1.1. Skip Frames*

As mentioned in the paper, we skip a set number of frames, as it is unnecessary to take actions for each step through the environment. In our experiments, we have set the skip factor to 4, deterministically. The observation that is returned after taking a step is equivalently the observation obtained after 4 consequent actions, and the reward obtained is the cumulative sum of the rewards obtained over the last 4 steps.

#### *9.1.2. Image Processing*

In the learning process, colors do not affect the training of the agent, all the more, it affects the computation load negatively. Thus we convert the RGB image into a Grayscale image using the following expression,

$$img_{gray} = 0.299 * img_{color}[\text{red}] + 0.587 * img_{color}[\text{green }] + 0.114 * img_{color}[\text{blue}]$$

The result of this transformation is also called the Luminosity channel. Post this, the required size of the image, cropping the score/lives information is obtained by accordingly selecting the range of pixels that are relevant to the training process (this process is done separately for each environment). The image is then resized to $80 \times 80$.

*9.1.3. Scaling and Buffer*

The image now has values ranging from [0, 255]. These values are scaled to [0,1] for proper training of the Convolutional Network. The observations are then added to a buffer and returned to the emulator.

### *9.2. Architecture for Atari tasks*

The DQN architecture for Atari tasks involves two neural networks, the training and target networks, both having the same architecture.

In our experiments, we have maintained consistency with the paper [3], in terms of both architecture as well as the hyperparameters used. The inputs given to the network have shape $(80 \times 80 \times 4)$, as produced by the wrapper on the emulator.

The first hidden layer convolves 32 filters of $8 \times 8$ with stride 4 with the input image and applies a rectifier non-linearity (ReLU). The second hidden layer convolves 64 filters of $4 \times 4$ with stride 2, again followed by a ReLU layer. This is followed by a third convolutional layer that convolves 64 filters of $3 \times 3$ with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. All the networks for the simple task were built using `Keras` module

As before, the Dueling architecture requires the splitting of the layers after the Convolutional layers. We used a single unit Dense layer for the Value stream and then used `expand_dims` to obtain the same value across all actions. For the advantage stream, we used a Dense Layer having units equal to the number of actions. These two layers' outputs were added using `Lambda` layers.

| **Hyperparameters** | **Value** | **Description** |
|---|---|---|
| batch_size | 32 | Number of training cases for every gradient descent update |
| replay_memory_size | 25000 | total size of the memory buffer |
| history_depth | 4 | number of observations coupled together as one input to the network |
| target_update_freq | 10000 | number of steps after which the target is made equivalent to the training network |
| discount_factor | 0.99 | discount factor $\gamma$ in Q-Learning update |
| action_repeat | 4 | Repeat each action selected by the agent this many times |
| learning_rate | 0.0001 | Learning rate used by Adam/RMSProp |
| initial_epsilon | 1 | Initial value of $\epsilon$ in exploration |
| epsilon_decay | 0.00001 | decay value of $\epsilon$ |
| final_epsilon | 0.02 | Final value of $\epsilon$ in the exploration scheme |
| replay_start | 32 | The number of steps before sampling from the replay buffer |

**Table 2.** Hyperparameters for Atari tasks

### *9.3. Results*

| Algo | Model | $\mu$-Absolute | $\sigma$-Absolute | $\mu$-$\epsilon$-greedy | $\sigma$-$\epsilon$-greedy |
|---|---|---|---|---|---|
| dqn | initial | -21.0 | 0.0 | -21.0 | 0.0 |
| | final | 18.83 | 2.936 | 18.97 | 2.874 |
| | best | 18.65 | 2.347 | 18.75 | 2.754 |
| double-dqn | initial | -21.0 | 0.0 | -21.0 | 0.0 |
| | final | 19.42 | 4.224 | 19.23 | 5.122 |
| | best | 16.41 | 10.905 | 16.10 | 10.232 |
| dueling-dqn | initial | -20.22 | 0.934 | -21.0 | 0.0 |
| | final | 20.44 | 0.875 | 20.23 | 0.924 |
| | best | 20.61 | 0.763 | 20.47 | 0.852 |

**Table 3.** Atari Pong: Results of saved models

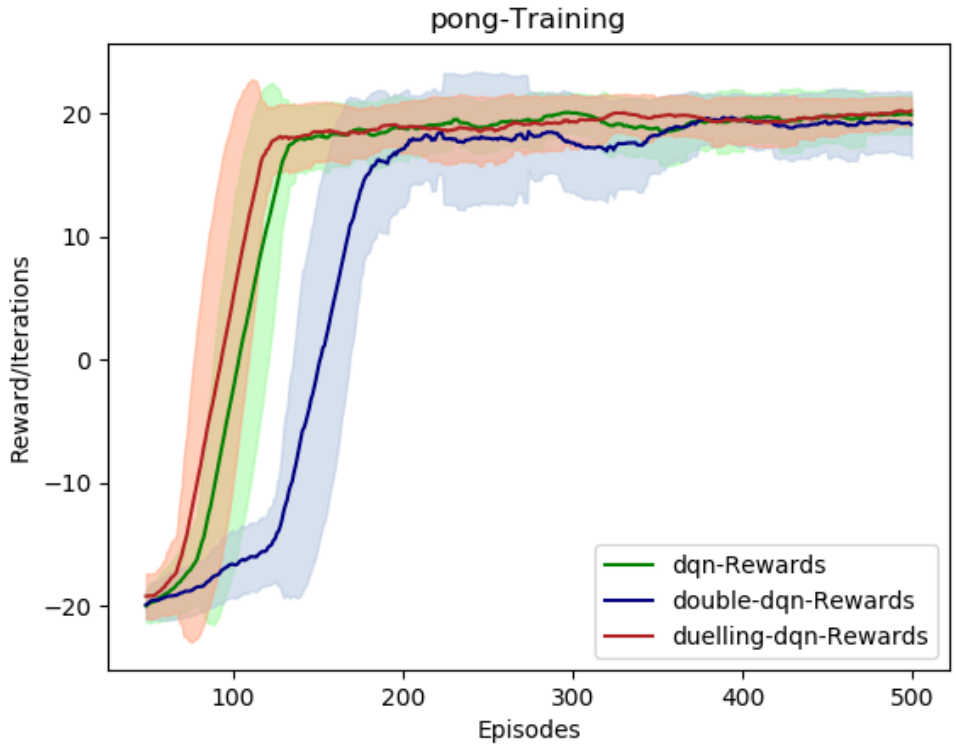



**Figure 4.** Training Graph depicting training rewards of the Pong environment (Atari Environment)

The values obtained in our experiments match the values presented in [3], [5], [6]. The training graphs along with the statistics from the frozen model suggests that the Dueling-DQN version performs the best, in terms of all the following metrics:

- Training Speed - Dueling-DQN learns and moves towards the optimal strategy the fastest

- As the model trains for sufficient amount of time, it has the least variability in terms of rewards generated and number of steps taken for completion
- The frozen models are more stable as in the variance produced is the minimum in the 'best' model chosen.

## 10. Generative Adversarial Networks

### *10.1. Introduction*

Generative Adversarial Networks (GANs) were first introduced as a Deep-Learning solution to the problem of learning complex probability distributions and computations that arise in Maximum Likelihood Estimates and similar strategies. A generative model includes the distribution of the data itself, and is capable of telling us how likely a given sample is. The basic task of the generative model that we are training is to produce convincing 'fake' data that looks like it is drawn from the original dataset. [9]

GANs are a clever way to train an unsupervised learning problem as a supervised learning problem with two sub-units.

### *10.2. Overview of the GAN structure*

A GAN has two parts:

(1) **The Generator** which generates plausible data. The Generator takes fixed length random vectors as inputs to produce samples in the domain of the problem. The random vector is typically drawn from a Gaussian Normal Distribution, to seed the Generative Process. The vector space is known as the latent space $\mathbb{Z}$, comprised of latent variables.

Random Input Vector → Generator Model → Generated Example

(2) **The Discriminator** takes an example from the domain as input (either real, or generated) and predicts a binary label (real, or fake). The Discriminator learns to distinguish between real and the Generator's fake data. The Discriminator penalizes the Generator highly if the Generator produces implausible data. The Discriminator is in essence a binary classifier, thus it could any network architecture applicable for a classification task (eg. Perceptron).

Real and Fake Training Data → Discriminator Model → Binary Classifier Outputs

### *10.3. GANs as a Two-Player Game*

GANs are based on a `Minimax` game rather than an optimization [9], and have a value function that one agent seeks to maximize and the other seeks to minimize. The game terminates at a saddle point that is a minimum with respect to one players strategy and a maximum with respect to the other players strategy.. The generator network directly produces samples. Its adversary, the discriminator network, attempts

to distinguish between samples drawn from the training data and samples drawn from the generator.

The two models, the Generator and the Discriminator are trained together. The Generator generates a batch of samples in the domain. The Discriminator is fed these samples along with an equal number of real samples from the domain, to be classified as real or fake.

The Generator is trained to become better at generating more plausible samples, in other words, fool the Discriminator. The Discriminator is trained and updated to get better at classifying samples. In this way, the Generator and Discriminator are playing a zero-sum game, in an adversarial sense, as they are competing against each other.

In an ideal scenario where we have a perfect Generator, we should see a prediction probability of 50%, which implies that the Discriminator is unsure (maybe real or fake, with equal probability). Once we have a Generator that has achieved this (or approximately close to this scenario), we can discard the Discriminator and use the Generator to independently generate samples from latent space noise.

### *10.4. Loss Function*

As GANs are trained to generate probability distributions, we need to use loss functions that measure the distance between the actual probability distribution and the one generated. The most important loss function is the `Minimax` loss that was proposed in the paper that introduced GANs [10].

The generator's distribution $p_g$ over the data $x$, we define a prior on the input latent variable $p_z(z)$. We can then represent $G(z;\theta_g)$ as the function represented by the Generator network with parameters $\theta_g$. We define another function $D(x;\theta_d)$ as the Discriminator network which outputs the probability that $x$ is generated from the data rather from $p_g$. The whole network can be defined as a minimax value function $V(D,G)$,

$$\min_G \max_D V(D,G) = \mathbb{E}_x\Big[logD(x)\Big] + \mathbb{E}_z\Big[log(1-D(G(z)))\Big]$$

This formula basically is the cross-entropy between the real and the generated distributions. Thus, in our implementations we can use the `binary-cross-entropy` loss function.

Since the generator can not access the $\mathbb{E}_x\Big[logD(x)\Big]$, the generator only uses the $\mathbb{E}_z\Big[log(1-D(G(z)))\Big]$ as its loss function.

The $\mathbb{E}_z\Big[log(1-D(G(z)))\Big]$ is seen to cause the network to get stuck at local minima. Thus we modify the Generator loss to maximize $\mathbb{E}_z\Big[log(D(G(z)))\Big]$ instead.

### *10.5. Training Process*

The GAN training process is not very straightforward. The combined GAN network has to separately handle the training of both Generator and Discriminator in an asyn-

chronous way. This is done in the following way:

(1) The Discriminator is trained separately, by giving it equal samples from a frozen Generator and the real data source. The losses are calculated and backpropagated only through the Discriminator network.
(2) The Discriminator is then frozen, and the Generator is trained. The loss obtained from the Discriminator is backpropagated to train only the weights of the Generator.

These steps are alternated until convergence. Convergence can be obtained when the Discriminator performance nears 0.5. This means that the Discriminator can successfully classify only half of the samples. In other words, it is completely unsure about the source of the sample.

#### *10.5.1. DCGANs*

DCGANs (Deep Convolutional GANs) are the most common form of GANs which find its relevance mainly in the field of Computer Vision, and Image Generation. Here the Generator is a Deconvolutional Neural Network with the capacity to generate the original picture resolution from data, while the Discriminator is a Convolutional Neural Network which extracts features from the Image.

## 11. Analysis of GANs to generate simple distributions

As an initial analysis and introduction to the field of GANs, we implemented a simple version of GAN, which would learn simple distributions like `Standard Normal`, `Exponential` and `Beta`. We look at the pseudocode of the algorithm implemented as presented by *Ian Goodfellow et. al* [11].

### *11.1. Model Architecture*

(1) The generator is a 4-Layer Fully Connected Neural Network with 16 units each, with a LeakyReLU activation function. The input layer has latent_dimension number of units, and has a single unit linear activated unit as output
(2) The Discriminator is a general classifier, with 2 Fully-Connected Layers with 64 units each, with a LeakyReLU activation function. The input is 1 unit in our case (uni-dimensional data) and the output is a single-unit with Sigmoid activation, for binary classification.

Both models are compiled with `Adam` optimizer, with `learning_rate`=0.001, with `loss=binary crossentropy`

### *11.2. Algorithm*

**Algorithm 3:** Simple GAN for generating 1-D distributions

**for** *number of training iterations* **do**
  **for** *k steps* **do**
    Sample mini-batch of $m$ noise samples $z_1, z_2, ..., z_m$ from prior $p_z(z)$;
    Sample mini-batch of $m$ real-world examples $x_1, x_2, ..., x_m$ from prior $p_{data}(x)$;
    Update the Discriminator model with a Stochastic Gradient step:;

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left( x^{(i)} \right) + \log \left( 1 - D\left( G(z^{(i)}) \right) \right) \right]$$

  **end**
  Sample mini-batch of $m$ noise samples $z_1, z_2, ..., z_m$ from prior $p_z(z)$;
  Update the Discriminator model with a Stochastic Gradient step:;

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[ \log \left( 1 - D\left( G(z^{(i)}) \right) \right) \right]$$

**end**

### *11.3. Implementation Details*

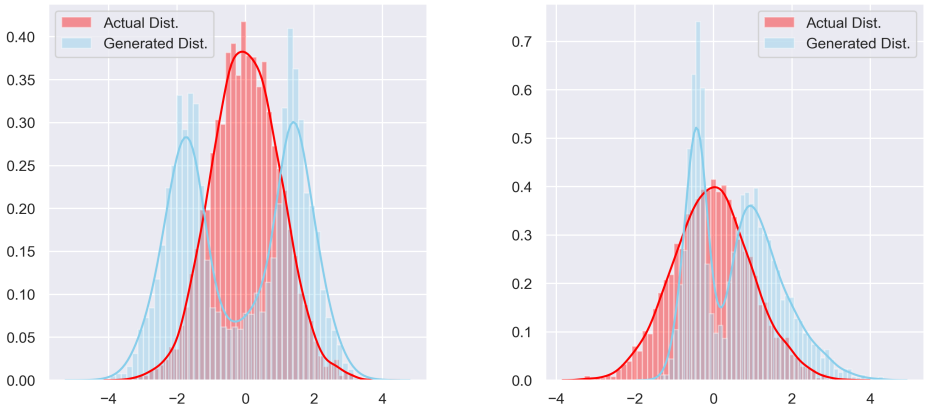
The noise samples are generated by a prior distribution $p_z(z)$, which is generally Gaussian Normal with zero mean and unit variance. The real samples in our implementation belong to either Standard Normal, Exponential or Beta distributions.

Each iteration of the algorithm presented above is run for 30,000 iterations across 600 curated batches. Each training batch consists of $batch_size/2$ real and fake samples. The Discriminator is trained on both the real and fake samples. The Discriminator is then made un-trainable, in order to freeze the weights so that the Generator (GAN model) can be trained on the noise samples.

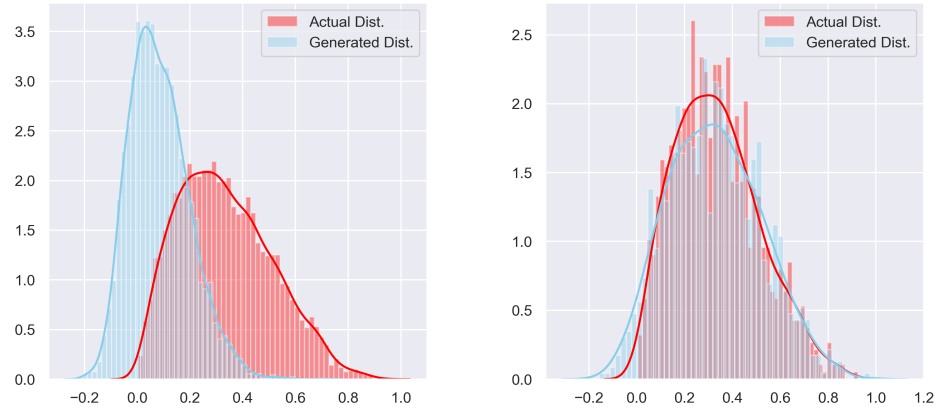
The losses and accuracies of the Generator and Discriminator models are saved. The best Generator model is also saved for later generating the test batch of the samples.

### *11.4. Results*

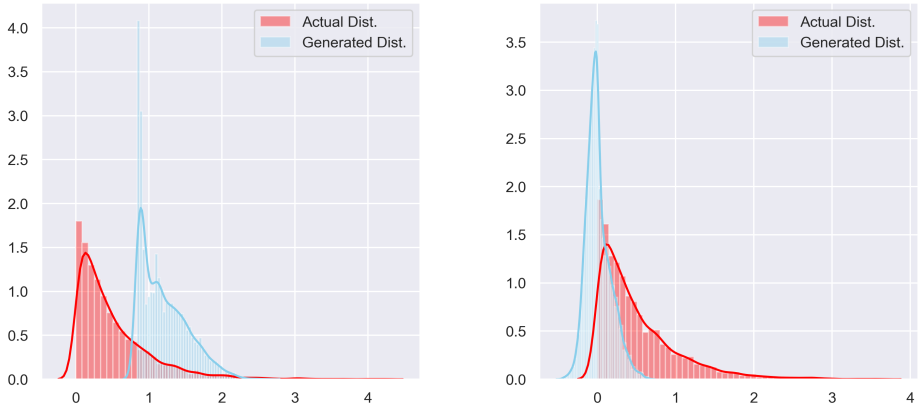
We used the saved training Generator models and gave it noise as input. The models generated samples according to the probability distributions of the training data. Standard Histogram plots were used to visualize the probability distribution of the resulting data.




(a) Normal Distribution, $\mu = 0, \sigma = 1$ : Left: Untrained Model, Right: Trained Model




(b) Beta Distribution, $\alpha = 2, \beta = 4$ : Left: Untrained Model, Right: Trained Model




(c) Exponential Distribution, $\lambda = 2$ : Left: Untrained Model, Right: Trained Model

**Figure 5.** Graphs showing the generation of simple 1-Dimensional Probability Distributions from data samples using GANs

## 12. Analysis of GANs on Images

The most interesting applications of GANs are seen in the world of Computer Vision and Image Generation. It is eerie how GANs can learn how to process and generate real-life images. In this section we explore the capability of GANs to learn complex probability distributions that correspond to the image domain. We will train two networks one each on Vanilla GANs, and another on Conditional GANs, and explore how both can be used in our problem domain in RL.

### *12.1. Training Vanilla GANs to generate MNIST digits*

We have trained a Vanilla GAN to handle the image processing and generate MNIST digits based on the training images provided to the network. In this GAN we do not provide the GAN any special information about the class, but give the network a shuffled dataset, from which it understands the innate distribution. From a completely trained model, we are able to generate legible digits which can be part of the MNIST dataset.

#### *12.1.1. Network architecture to handle images*

Developing a GAN for generating images requires both a discriminator convolutional neural network model for classifying whether a given image is real or generated and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values.

(1) The Generator is a complex system, since it has to create an image from noise in 1-D. The latent space is a 100-Dimensional vector. The first is a Dense layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. In a normal CNN, we have multiple layers learning parallel feature maps simultaneously. We need to reverse the same process, as the Generator needs to now create/invent images. Therefore, the first hidden layer, the Dense layer needs enough nodes for multiple low-resolution versions of our output image, such as 128. Thus we have a Dense Layer with shape $(7 \times 7 \times 128)$.

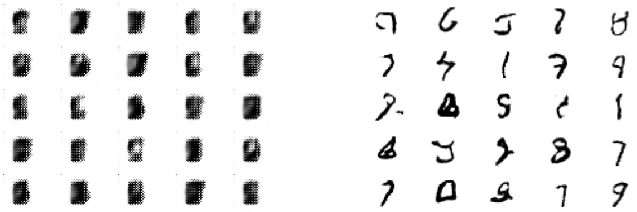
Post this, we need an `UpSampler` and an inverse `Conv2D` layer, this is handled by a `Conv2DTranspose` layer. This can be repeated until we arrive at $28 \times 28$.

(2) The Discriminator model has two convolutional layers with 64 filters each, a small kernel size of 3, and larger than normal stride of 2. The model has a single node in the output layer with the `Sigmoid` activation function to predict whether the input sample is real or fake. The model is trained to minimize the `Binary Cross-Entropy` loss function.

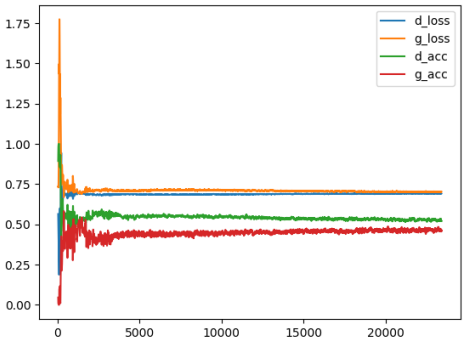
#### *12.1.2. Implementation Details and Results*

The training process is exactly the same as that presented in section. Certain Code snippets were obtained from [12].

The figures here present the images that were generated by the model initially before training, and after convergence (best model).



(a) Left: Untrained Model, Right: Trained Model




(b) Generator and Discriminator Loss and Accuracy

**Figure 6.** MNIST GAN

### *12.2. Training Conditional GANs to generate Class-Specific Images*

Conditional GANs were born out of two basic needs in areas that the Vanilla GAN was lacking, by using class-labels and making use of data distributions specific to classes.

(1) **Improve the GAN** By using class-labels, we can focus the GANs learning towards the data of a particular type, and improve the stability of the model.
(2) **Targetted Generation** Class-labels can be used for targetted generation of images from a particular class, something which cannot be controlled in Vanilla GAN.

Though there is an option to map the relation between the latent space and the output images, this relation is very complex, and not pragmatic. In this section we condition the GAN with additional information, to train the network to learn conditional probability distributions.

#### *12.2.1. Modified Architecture*

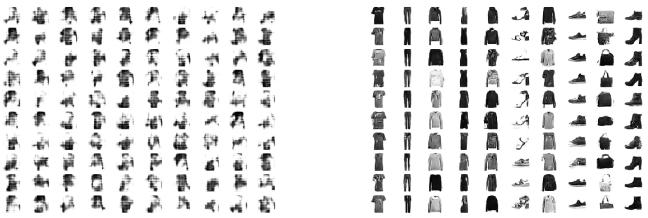
The architecture of a typical DCGAN is now modified to take in extra input of the class label. The class label is given as an additional input to both the Generator as well as the Discriminator. Though there are multiple ways to include the class label into the network, the best practice is to use an Embedding layer (this layer creates a unique representation for each class-label) of a small size (less than the dimension of the input). This is then up-sampled and reshaped to a single plane feature map(with the same dimension as the data stream), which is then concatenated to the original

data stream.

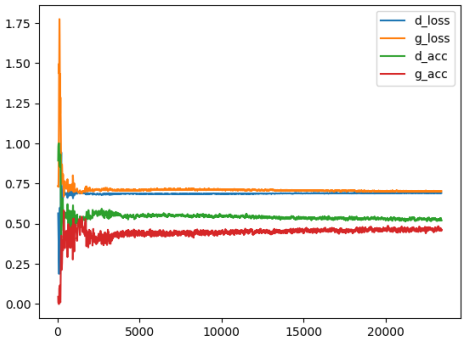
This change is done in both the Generator and the Discriminator networks.

### *12.2.2. Results*

In this section, we present the images that were generated specific to the classes. We can see the separate class images in each column. Code for this implementation was obtained from [13].



(a) Left: Untrained Model, Right: Trained Model




(b) Generator and Discriminator Loss and Accuracy

**Figure 7.** MNIST Fashion cGAN

## 13. Generation of MDPs using GANs

In this section we present a novel method of the simulation of a Markov Decision Process (subsection 2.2) using the GAN as the simulator/emulator. Due to constraints of resources, as a proof of concept we are presenting a small-scale implementation of the same, having 4 states $S = \{S_0, S_1, S_2, S_3\}$, with transitions based on the transition probabilities matrix of the system $P$.

The State-Transition Matrix of the Markov Decision Process considered in our experiments is as below.

| | $S_0$ | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|
| $S_0$ | 0.6 | 0.1 | 0.2 | 0.1 |
| $S_1$ | 0.2 | 0.5 | 0.2 | 0.1 |
| $S_2$ | 0.2 | 0.0 | 0.6 | 0.2 |
| $S_3$ | 0.1 | 0.1 | 0.1 | 0.7 |

The table can be interpreted in the following way: The $(i, j)$ entry of the matrix is the probability of moving from state $S_i$ to state $S_j$, and is equal to $P_{S_i,S_j}$. In our analysis, we try to recreate this transition matrix with the help of our trained GAN.

### *13.1. Implementation and Architecture*

In our experiments, we have considered 4 states. We train a GAN for each state, to obtain the transition probabilities for each state. For each state $S_i$, we train the GAN to obtain the probabilities $P_{S_i,S_j}, S_j \in S$. We have excluded the notion of an action, but considered it intrinsically as being involved in the transition to the next state.

As our states are discrete, we have to come up with a representation for them, as GANs do not work very well with Discrete data. Thus, obtaining inspiration from cGANs, we implement a one-hot representation of the states, like $S_0 = \{1, 0, 0, 0\}$, $S_1 = \{0, 1, 0, 0\}$ and so on.

As in all GANs, we require a real-data batch and a fake-data batch. For the real-data, we generate samples based on the transition probabilities fixed. This batch of real-data (labeled as 1) is combined with the fake-data (labeled as 0) to train the Discriminator. The fake-data is generated by the Generator model.

The Discriminator remains to be a simple Classifier network, which takes a 4-unit input, and has two Fully Connected Layers with 32 units each. Each FC layer has a LeakyReLU activation, and the model is trained with an Adam optimizer with `learning_rate=0.0001` and `binary cross-entropy` loss.

The Generator takes in an input from the Latent Space which is picked from a 5-D Gaussian Normal Distribution with 0 mean and unit variance. This is followed by 4 Fully Connected Layers with 16 units each, with a LeakyReLU activation layer. The model is also trained with an Adam optimizer with `learning_rate=0.0001` and `binary cross-entropy` loss.

The best model is frozen and samples are generated from the Generator to test the probabilities of the states generated.

### *13.2. Results*

The transition probabilities obtained from the GANs almost matched the original transition probabilities of the model. Thus, we show that this is a valid method to generate transition probabilities of simple MDPs.

| | $S_0$ | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|
| $S_0$ | 0.6117 | 0.1052 | 0.2027 | 0.0754 |
| $S_1$ | 0.1818 | 0.5914 | 0.2168 | 0.01 |
| $S_2$ | 0.2472 | 0.0 | 0.7528 | 0.0 |
| $S_3$ | 0.0975 | 0.121 | 0.0 | 0.7812 |

## References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

[5] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[6] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[7] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[8] Phil Tabor. Deep Q-Learning from Paper to Code.

[9] Google Developers. Generative Adversarial Networks. `https://developers.google.com/machine-learning/gan`.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[11] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

[12] Jason Brownlee. How to Develop a GAN for Generating MNIST Handwritten Digits.

[13] Jason Brownlee. Develop a Conditional GAN (cGAN) From Scratch.