Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Investigating the Effects of Double Deep Q-Learning in Video Game Environment

Kanika Ghiloria

B.Tech (IT)

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Vincent Wade

Co-Supervisor: Hossein Javidnia

September 2020

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Kanika Ghiloria

September 7, 2020

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Kanika Ghiloria

September 7, 2020

Acknowledgments

I would like to express my gratitude to the following people without whom this dissertation would not have been a success.
First and foremost, I want to thank my supervisor Professor Vincent Wade for letting me be a part of such an amazing project and steering me towards this research area.
A heartfelt thanks to my co-supervisor Dr. Hossein Javidnia for his constant guidance, support and patience throughout this dissertation.
A big thanks to my family for their everlasting support and faith in me.
Lastly, I would like to thank my friends for their help and guidance whenever I needed it.

Kanika Ghiloria

University of Dublin, Trinity College
September 2020

Investigating the Effects of Double Deep Q-Learning in Video Game Environment

Kanika Ghiloria, Master of Science in Computer Science

University of Dublin, Trinity College, 2020

Supervisor: Vincent Wade

Over the past decade, the advancement of Deep Reinforcement Learning (DRL) has shown a great potential towards creating autonomous systems capable of understanding the surrounding world without supervision. This dissertation aims at examining the performance of Double Deep Q-Learning (DDQN) and Deep Q-Learning (DQN) techniques with different epsilon decay strategies on the learning and performance of a RL agent in a video gaming environment. Three AI agents including Deep-Q Network (baseline model) using decaying epsilon greedy strategy, Backward Q-Learning using greedy approach and Double Deep Q-Learning using sinusoidal exploration decay are trained and evaluated in different environments of a popular arcade-style game called Flappy Bird. The performance of the agents are measured using standard metrics such as maximum/average score and number of training iterations to achieve the best score.

Summary

The advancement of Reinforcement Learning (RL) techniques over the past decade has shown a significant potential in a wide range of applications such as robotic, industrial automation and business strategy planning. Over the past couple of years, video games have become a popular testing platform for efficiency analysis of such systems. The similarities (such as passing the obstacles, avoiding negative actions and adapting to new challenges and environments) between the real world and the video game environments have made video games a prevalent choice as test-beds.

This dissertation focuses on the ongoing research in Deep Reinforcement Learning (DRL), specifically towards the model free learning algorithms. It begins with an introduction to the general field of RL followed by a review of some of the state of the art DRL algorithms.

An experimental study is further conducted to investigate the effects of different DRL algorithms and exploration - exploitation balancing techniques in a state-of-the-art video game environment called Flappy Bird. Three DRL algorithms with different exploration decaying strategies are used to create the following AI agents:

- DQN agent (Deep Q-Learning with decaying epsilon greedy strategy).
- Q-Learning agent (Q-Learning with greedy approach).
- DDQN agent (Double Deep Q-Learning with sinusoidal exploration decay strategy).

The training and testing environments are designed based on four models with varying difficulty levels. The DRL agents are trained in the training environment and their performance is evaluated in both training and testing environments based on three performance criteria: Maximum Score, Average Score and Number of Training Iterations. The main goal is to examine the effects of DDQN algorithm with Sinusoidal exploration on the performance of the agent compared to that of standard DQN algorithm with Decaying epsilon greedy strategy.

A comparative study between the agents' performance concluded that the DDQN agent converged to the optimal policy much faster and achieved higher scores an all the environments.

The implementations of the networks and models discussed in this thesis can be found at: `https://github.com/kanikaghiloria/DeepRL_DQN_DDQN/`

Contents

Acknowledgments **iii**

Abstract **iv**

Summary **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Reinforcement Learning 1

1.2 Thesis Objective and Overview 3

1.3 Thesis Structure 3

Chapter 2 A Deeper Look into Reinforcement Learning **5**

2.1 Markov Decision Processes 6

2.1.1 The Dynamics of MDP 7

2.1.2 Transition Function 8

2.1.3 Reward Function 8

2.1.4 Goal 8

2.1.5 Discounting 9

2.1.6 Value Function 9

2.1.7 Q-Value Function 10

2.1.8 Bellman Optimality Equation 10

2.2 Offline vs Online Learning 10

2.3 Off-policy vs On-policy Learning . . . 11
2.4 Exploration vs Exploitation Dilemma . . . 11
2.4.1 Greedy Approach . . . 11
2.4.2 Epsilon Greedy Strategy . . . 12
2.4.3 Decaying Epsilon Greedy Strategy . . . 12
2.4.4 Sinusoidal Exploration Decay Strategy . . . 13
2.5 Q-Learning . . . 13
2.6 Double Q – Learning . . . 15
2.7 Deep Reinforcement Learning . . . 16
2.7.1 Deep-Q Learning . . . 16
2.8 Challenges in RL . . . 17

Chapter 3 Literature Review **18**
3.1 History of Reinforcement Learning in Gaming . . . 20
3.1.1 TD – Gammon . . . 20
3.1.2 AlphaGo . . . 21
3.1.3 DOTA 2 . . . 21

Chapter 4 Flappy Bird Game **22**
4.1 Difficulty Levels . . . 23
4.2 Train and Test environments . . . 24
4.3 Action Space . . . 26
4.4 State Space . . . 26
4.5 Goals and Rewards . . . 27
4.6 Implementation Details . . . 27
4.6.1 Input Pre – processing . . . 27
4.6.2 Convolutional Neural Network Architecture . . . 27
4.6.3 Software and Hardware . . . 28

Chapter 5 Experiments **30**
5.1 DQN Agent – Baseline Model . . . 30
5.2 Q-Learning Agent . . . 33
5.3 DDQN Agent . . . 35

Chapter 6 Results and Analysis **38**
6.1 Sinusoidal Exploration Decay vs Epsilon Greedy Decay 41
6.2 Discussion 43

Chapter 7 Conclusion **48**
7.1 Future Work 49
7.1.1 Offline Learning in DRL 50

Bibliography **51**

List of Tables

4.1 Hardware specifications 28
4.2 Software specifications 29

5.1 DQN parameters 33
5.2 DDQN parameters 37

6.1 DQN agent results - training environment 39
6.2 DDQN agent results - training environment 39
6.3 DQN agent results - testing environment 40
6.4 DDQN agent results - testing environment 40

List of Figures

1.1 Flow of the project 4

2.1 Agent's interaction with the environment 6

4.1 Flappy Bird game - Graphical user interface 23
4.2 Easy model 25
4.3 Moderate model 25
4.4 Moderate model 25
4.5 Moderate model 25
4.6 Training environment - Hard model 26
4.7 Testing environment - Hard model 26
4.8 CNN architecture used in DRL algorithms 28

5.1 RL agents 31

6.1 Performance plot - Easy model 42
6.2 Performance plot - Moderate model 42
6.3 Performance plot - Moderate model 42
6.4 Performance plot - Moderate model 42
6.5 Training Iterations plot 43
6.6 Epsilon vs Training Iterations plot for DQN agent - General model . . 44
6.7 DQN agent's Score vs Training Iterations plot - General model 44
6.8 Epsilon vs Training Iterations plot for DQN agent - General model . . 45
6.9 DDQN agent's Score vs Training Iterations plot - General model 45
6.10 Score vs Training Iterations plot - comparison 46

Chapter 1

Introduction

Creating AI systems that have the ability to learn from their environment and interact with it, has always been thought-provoking. These systems can range from robots to software-based agents. An AI should fundamentally "understand the world around us" [1]. Video games have proven to be amazing test-beds for AI algorithms. Ability of the AI agent to interact with the controlled game environment and solve complex problems make these games a popular domain for AI research. These simulated environments are often created for a specific problem, and can be used to prove or disprove the hypothesis quickly. This dissertation aims at investigating the effects of Double Deep Q-Learning (DDQN) [2] and different exploration decay strategies such as Exponential and Sinusoidal [3] in a state of the art video game environment known as Flappy Bird. The agent is trained in different scenarios with varying difficulty levels to examine its performance with regards to the variation in environment. The preliminary evaluations indicate a significant improvement in the performance of the agent while being trained using the new decay strategy.

1.1 Reinforcement Learning

Reinforcement Learning (RL) in one of the most cutting-edge approaches that focuses on developing AI systems. RL is more of a goal-directed learning technique from interacting with the environment when compared to supervised and unsupervised machine learning. An agent is designed to interact with its environment and learn to take ap-

propriate actions respectively i.e. learn from its own experience. Agent receives an immediate reward signal (positive or negative) from the environment after each action. The ultimate goal of the agent is to maximize this cumulative numerical reward. The actions taken by the agent could impact not only the immediate reward, but also the subsequent rewards. 'Trial and error search' and 'delayed reward' are the most important distinguishing features of RL [4]. One of the main challenges in RL is the dilemma of exploration vs exploitation. Exploitation entails the agent selecting one of the actions it has tried in the past, which will lead to a definite positive reward. Exploration on the other hand involves agent trying random actions for the same situation in order to understand which one of them will lead to the maximum reward. The agent might exploit its existing knowledge and lose an opportunity to discover an even better action. Or else, it can explore more actions in hope of finding better options and risk getting a negative reward. Ideally, it is expected that the agent will need to explore more in the beginning of the training and eventually reduce the exploration.

Following four elements of a RL system can be identified apart from the agent and environment [4].

Policy – It guides an agent to take specific actions in perceived states. It is a set of state – action rules and defines the behaviour of an agent. A policy may be either stochastic or deterministic in nature.

Reward Signal – It defines the goal of Reinforcement Learning. Agent receives an immediate numerical reward signal after each action which is determined by the state in ended up in. This signal therefore defines the good and bad actions. These signals tend to alter the policy if the defined action for any state leads to a low or better reward.

Value Function – This function determines the impact of any action not just on the resulting state, but on all the upcoming states. Values of any state is equivalent to the total reward that the agent will accumulate in the future. This value is estimated by the agent using the reward it receives from the environment.

Model of environment – It is an optional element and the methods that use it are called model-based methods while the rest are called model-free methods. A model

specifies the behaviour of the environment and therefore learning is not completely based on trial and error search.

1.2 Thesis Objective and Overview

The dissertation focuses on examining the effects of DDQN technique with different epsilon decay strategies on the learning and performance of RL agent in a video gaming environment. The overall objectives of the project are listed below:

1. Study the state-of-the-art Deep Reinforcement Learning (DRL) algorithms.
2. Analysis of a Deep Q-Learning (DQN) technique with exponential epsilon decay trained in an Atari gaming environment.
3. Exploring DDQN algorithm with sinusoidal epsilon decay strategy in a gaming environment.
4. A comparative study on the implemented algorithms and strategies used based on their performance and training time.

Figure 1.1 shows the approach that was followed for the implementation and evaluation in this dissertation.

1.3 Thesis Structure

The rest of this dissertation is structured as follows:

Chapter 2 presents a detailed description of Reinforcement Learning and its components. This section explains the Markov Decision Processes, types of learning, various RL strategies that are being used for better training and the description of algorithms used in this dissertation.

Chapter 3 presents the review on some of the most significant state-of-the-art methods and achievements in Reinforcement Learning.

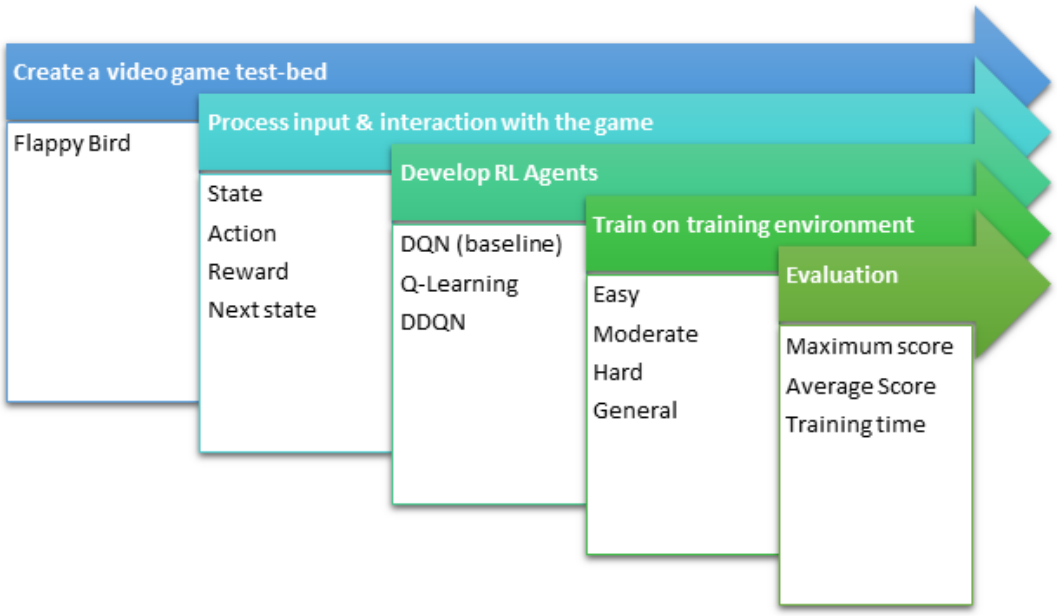

Figure 1.1: Flow of the project

Chapter 4 describes the implementation details and components of the video game environment – Flappy Bird created for this dissertation.

Chapter 5 presents the detailed description of the experiments conducted to measure the performance of different RL agents. Implementation details and description of DQN agent, Q-Learning agent and DDQN agent is presented in detail.

Chapter 6 constitutes the results obtained from the experiments in the previous section and their analysis.

Chapter 7 concludes the dissertation with the learning outcomes, possible areas for future research and concluding remarks.

Chapter 2

A Deeper Look into Reinforcement Learning

Reinforcement Learning constitutes learning from interaction and how to act to achieve defined goals. The RL agent (everything is known and controllable) interacts with the environment (may or may not be known) over a sequence of discrete time steps t. Agents objective is to maximise the cumulative numerical reward that it receives for each interaction, while following a defined policy. The interaction of an agent with its environment can be visualised from Figure 2.1 where,

- s_t –current state (representation) of the environment at time step t, where $s_t \epsilon S$ (a set of valid states in the environment)
- a_t – action selected by the agent for the state s_t, where $a_t \epsilon A$ (a set of valid actions for state s_t)
- s_{t+1} – state of the environment at time step $t + 1$, where $s_{t+1} \epsilon S$
- r_t – numerical reward received by the agent as a consequence of action a_t, where $r_{t+1} \epsilon R$ (a set of numerical rewards)

The agent receives the information regarding the current state of the agent from the environment, chooses the best possible action and takes it. The environment receives this information and returns the numerical reward / penalty and information of the

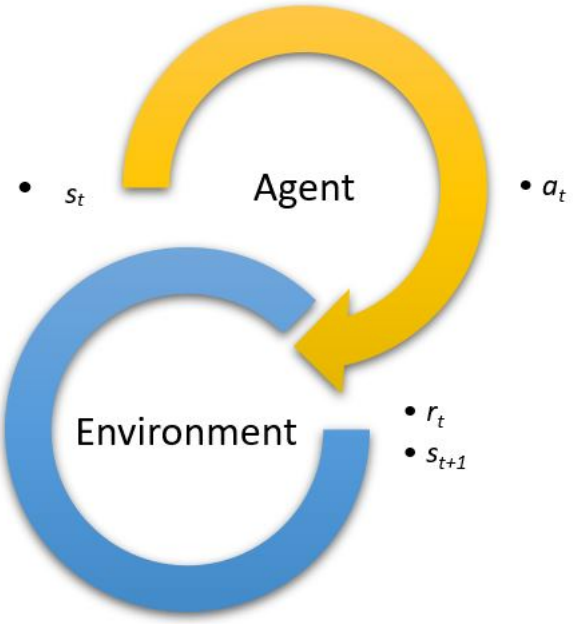

Figure 2.1: Agent's interaction with the environment

new state as the outcome. This process is repeated until the agent learns to achieve the defined goals.

RL problems can be formulated in mathematical form using Markov Decision Processes (MDPs). They are a formalization of sequential decision making where the actions impact both immediate rewards along with subsequent situations and future rewards (delayed rewards) [4].

2.1 Markov Decision Processes

Markov Decision Processes (MDP) frame the problem of agent's learning from interacting with the environment to achieve a goal. The learner is called an agent while the interaction takes place in an environment. The agent selects an action (from a defined set of actions) to interact with the environment. The environment responds with a numerical reward and a new situation for the agent to tackle. This process goes on in a continuous loop with the ultimate goal to maximise the cumulative reward over time. The MDP framework proposes that any problem of goal-directed learning

from interaction can be reduced to three signals passing between the agent and the environment: actions, states and awards. The set values for current state (s), action (a), reward (r) and next state (s,) constitute as experience for the agent. The primary aim of the MDP is to find an optimum policy function for the agent.

The challenge in practical implementations is that, despite having the complete model of the environment and the optimal policy, agent fails to perform enough computation for each timestep. Also, availability of large memory is required for storing the computations and complete state space. Therefore, the agent uses approximations instead to achieve the desired goals.

2.1.1 The Dynamics of MDP

In MDP, the finite variables r_t and s_t have a well-defined probability distribution which only depends on its previous state and action. This implies that for some random variables, $s' \in S$ and $r \in R$ there is a probability of these values occurring if the particular values of previous state and action are present.

$$p\left(s', r|s, a\right) = Pr\left(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right),\ \forall\ s, s' \in \mathcal{S},\ r \in \mathcal{R},\ a \in \mathcal{A}\left(s\right)$$

The dynamics of the environment [4] is characterised using the above equation which highlights that the reward and state at any particular time depends on its previous state and action.

The dynamic function p is a four-argument deterministic function represented by $S\ \times\ R\ \times S\ \times\ A\ \rightarrow\ [0, 1]$. p here gives the probability distribution of each s and a. This means that the probabilities of the all the values of states and rewards are dependent only on its previous state and action. This implies that the current state must contain all the information of the previous interactions of the agent with the environment. If this statement holds true, then the agent is said to have the markov property [4]. Using this dynamics function, all the information of the environment can be computed.

2.1.2 Transition Function

State transition probability is a three-argument function ($p : S \times S \times A \rightarrow [0, 1]$) [4] defined as :

$$p(s',\ r|s,\ a)\ =\ \sum_{r\ \in\ R} p(s',\ r|s,\ a),\ \ \forall\ \ s,\ s'\ \in\ S,\ \ a\ \in\ A,\ \ r\ \in\ R$$

It gives the probability that by selecting an action a in a state s at a given time t, the agent will arrive at the next state s' and receive the reward r. MDPs assume that the state transitions depend only on the last state of the agent and not on any of the previous states or actions.

2.1.3 Reward Function

Expected reward can be represented as a three-argument function (state – action – next state) [4] $r : S \times A \times S \rightarrow R$,

$$r(s, a, s') = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)},\ \forall\ s, s' \in S,\ a \in A,\ r \in R$$

Reward function determines the rewards using action taken in any particular state and the state agent has ended up in. Hence, this function maps the rewards / penalties to the observations. Reward function has a direct impact in altering the policy.

2.1.4 Goal

The agent receives a numerical reward from the environment as a consequence of each action taken for a particular state. The goal of the agent is to maximise the cumulative reward or expected return in the long run. Expected return is a function for future rewards that the agent trains to maximise. For every action that leads the agent towards a desirable or favourable state, it receives a positive numerical reward. For any action that leads the agent to any undesirable state, a negative reward or penalty is received. Hence, the rewards are setup to define the goals that agent is expected to receive.

2.1.5 Discounting

Actions taken by the agent may result in the positive reward after a number of time steps and states later. This is called as delayed reward [4]. Since the final goal is to maximise the cumulative reward, the agent will treat the actions leading to immediate rewards and the ones leading to rewards at a later stage as equal. To increase the value of immediate rewards over delayed rewards, discount rate γ (gamma) is introduced. Discount rate determines the current value of future rewards. Hence, the expected return is represented as:

$G_t \ = \ \sum_{k \ = \ 0}^{\infty} \gamma^k r_{t+k+1}, \ \ where \ 0 \ \leq \ \gamma \ \leq \ 1$

2.1.6 Value Function

Value function or state-value function [4] estimates how good it is for an agent to be in a particular state, which is defined in terms of the cumulative reward / expected return. These functions are defined using policies. A policy (π) is a stochastic set of rules that defines the mapping between the states and the probable actions. The RL agent learns from experience and modifies this policy eventually. The value function gives the expected return for each state s when agent is following a policy π.

$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r \ + \gamma v_\pi \left(s'\right)\right], \forall \ s \in S, a \in \ A, r \in R$

This equation is known as *Bellman equation* [4]. This equation denotes the association between the value of the state and its successor states.

Optimal Policy (π_*): Policy that is better than or equal to all other policies i.e. its expected return is greater than or equal to all the other policies [4]. There may be more than one optimal policy.

Optimum policies share the same *optimum state-value function* [4]. This function assigns to each state the maximum expected return that can be achieved by a policy. Optimum state-value functions are unique for any given MDP. It is formulated as:

$v_*(s) \ = max v_\pi(s), \forall \ s \in S$

2.1.7 Q-Value Function

Q-value function or *action-value function* [4] gives expected return for taking action a in state s under the policy π.

$q_\pi(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma v^*(s')], \forall s, s' \in S, a \in A, r \in R$

Where T represent the model of the environment.

Optimal policies share the common *optimal action-value function.* Optimal action-value function is the maximum achievable expected return for each state s when action a is taken by the agent while following policy π. It is represented as:

$q_*(s,a) \ = \ max \ q_\pi(s, \ a), \forall \ s \in \ S, \ a \in A$

2.1.8 Bellman Optimality Equation

Bellman optimality equation states that the value of a state following an optimal policy is equal to the expected return for the best action from that state[4]. Bellman optimality equation for $v*$,

$v_*(s) = maxq_{\pi*}(s,a) \Rightarrow v_*(s) = max \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')], \ \forall \ s \in S, \ a \in A, \ r \in \ R$

Bellman optimality equation for $q*$,

$q_*(s,a) \ = \sum_{s',r} p(s', r|s, a)[r + \gamma max q_*(s', a')], \ \forall \ s, s' \in \ S, \ a, a' \in A, \ r \in R$

2.2 Offline vs Online Learning

Offline Learning entails learning from the available and limited data of an environment. In *online Learning* on the other hand, along with learning the sequential decision-making tasks, agent has to explore the environment as well [5].

2.3 Off-policy vs On-policy Learning

On-policy learning agent [4] computes the Q-values for the states using the action mapped in its current policy. In this type of learning agent does not explore and keeps on following the policy. Whereas *Off-policy* learning [4] computes the Q-values for the states by taking different (or random) actions than the ones defined in the policy. Therefore, the agent follows a different policy to generate behavior (called *behavior policy*) and a different one to evaluate an improve (called *estimation policy*).

2.4 Exploration vs Exploitation Dilemma

Trade-off between exploration and exploitation is one of the challenges in RL. In order to increase the cumulative award, the RL agent needs to choose actions that has been tried in the past – Exploitation. Furthermore, for stochastic environments the agent has to try a particular action for a state multiple times to confirm its desirability. At any timestep for the explored state-space, there is at least one action whose value is greatest. These actions are known as *greedy actions*, which exploits the current knowledge of the agent. But to know the best action in any particular state (or situation) the agent needs to try actions whose outcome is unknown – Exploration. Even for an explored state-space, selection of nongreedy actions will lead to better estimation of their values. The dilemma is that the exploration and exploitation cannot be pursued exclusively if we need the algorithm to converge [4]. Exploitation (or greedy approach) lead to immediate high reward while exploration might result in low short-term reward and greater total reward in the long run. If the agent has many time steps left then exploring using non-greedy actions might be more fruitful in terms of achieving high cumulative reward. Actions that lead to negative rewards or undesirable state causes regret. Regret is the difference between the return of the optimal actions and the return of the action taken. The task of the RL agent is to minimize this total regret.

2.4.1 Greedy Approach

Here, the agent performs exploration and observe the rewards in the beginning. As the Q-function converges, the amount of exploration decreases. This approach selects

on one action that gives good results and keeps selecting it. This action might be good but not necessarily optimal. This approach exploits too much, does not balance the exploration and exploitation, increases cumulative regret and therefore greedy in nature.

2.4.2 Epsilon Greedy Strategy

Epsilon greedy strategy [4] is the simplest solution to the previous approach. This approach uses a hyper-parameter ε (epsilon), which is the probability of choosing between exploration and exploitation. The value of ε is the probability with which the agent takes random actions i.e. explores instead of exploiting. High value of epsilon would mean that agent takes more random actions and lower value would mean more greedy actions. Issue is that over time, the agent does not need to explore as much (the agent has been learning for a while and has found some optimal actions). The strategy keeps allocating fixed percentage of actions for exploration throughout the training and therefore ends up increasing the cumulative regret. This strategy explores too much.

2.4.3 Decaying Epsilon Greedy Strategy

To overcome the drawbacks of greedy and ε-greedy strategy, it is necessary to strike the correct balance between exploration and exploitation. In decaying ε-greedy strategy, the value of ε decreases exponentially or linearly over time and then settles to a fixed value. Hence, as the learning of the agent increases, the percentage allocated for the exploration decreases. This results into agent selecting completely random actions in the beginning of training (when value of ε is high), reducing the exploration over time (as ε decreases) and then settles down to a fixed exploration rate (when agent does not need to explore as much). Epsilon is updated at each time step using the following function:

$$\varepsilon = \varepsilon - \left(\frac{\varepsilon_o - \varepsilon_f}{X}\right)$$

Where,
ε_o: initial epsilon

ε_f: final epsilon
X: Total number of training time steps

2.4.4 Sinusoidal Exploration Decay Strategy

Since decaying epsilon greedy strategy is either linear or exponential in nature, it has been observed that the training curve of the agent flattens when epsilon decreases. Sinusoidal exploration decay strategy was proposed by R. Chuchro and D. Gupta [3] to handle this issue. It is a decaying function that exponentially decays over the episodes in sinusoidal manner. This method helps in escaping the local optima that agent might reach due to reduced ε.

Epsilon is updated at each timestep using the following equation:

$$\varepsilon = \varepsilon_o \bullet \varepsilon_d^x \bullet \frac{1}{2}\left(1 + cos\left(\frac{2\pi t}{X}\right)\right)$$

Where,
ε_o: initial epsilon
ε_d^x: decay rate
t: current timestep
X: Total number of training time steps

2.5 Q-Learning

Q-Learning [6] is a model-free algorithm. This implies that it estimates the optimal policy without the knowledge of the dynamics (transition and reward functions) of the environment. The aim is to learn the optimal policy by estimating the optimal action-value-function. Optimal policy can be represented as:

$$\pi\left(s\right) = argmax_a Q^*\left(s, a\right)$$

The Q-values of state-action pair are updated using following value iteration:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(R\left(s\right) + \gamma max_{a'} Q\left(s', a'\right) - Q_{old}(s,a)\right),\ \forall s, s' \in S,\ a, a' \in A,\ r \in$$

R

s: current state
s': next state
α: Learning rate
$R(s)$: Reward function
γ: Discount factor
$Q_{old}(s, a)$: estimated Q-value of the starting state
$max_{a'}Q(s', a')$: maximum estimated Q-value from next state

This equation shows iterative process of back-propagation which causes the learned action-value function to directly approximate the optimal action-value function. The algorithm uses a table to map all the states-action pairs to their current Q-values. Every iteration, the algorithm looks up the Q-values for the current state (and next state) from this table and updates them. Pseudocode for Q-Learning algorithm [**?**] is depicted in Algorithm 1.

Algorithm 1: Q-Learning

- Algorithm parameters: step size $\alpha \in (0, 1], small\ \varepsilon > 0$
- Initialise $Q(s, a), \forall s \in S, a \in A$, arbitrarily except that $Q(terminal, .) = 0$
- Loop for each episode:
  - Initialize S
  - Loop for each step of episode:
    - Choose a using policy derived from Q
    - Take action a, observer r, s'
    - $Q(s, a) \leftarrow Q(s,\ a) + \alpha\ (r + \gamma\ max_a Q(s',\ \ a) - Q(s,\ a)$
    - $s \leftarrow\ s'$
  - Until s is terminal

2.6 Double Q – Learning

One of the drawbacks of DQN is that it overestimates the Q-values leading to poor performance for some stochastic MDPs. Using maximum value to approximate the maximum expected value causes positive bias. To resolve this issue, Double Q-Learning [7] was proposed which uses a double estimator method that sometimes underestimates the maximum expected value instead of overestimating it. Double Q-Learning stores two Q functions and each is updated with the value of the other Q function for the next state. Algorithm 2 is the pseudo code for Double Q-Learning.

Algorithm 2: Double Q-Learnining

- Algorithm parameters: step size $\alpha \in (0, 1]\,, small\ \varepsilon > 0$
- Initialise Q^A, Q^B, arbitrarily except that $Q\,(terminal, .) = 0$
- Loop for each episode:
  - Initialize s
  - Loop for each step of episode:
    - Choose a, based on $Q^A\,(s, .)\, and Q^B\,(s, .)$
    - Take action a, observe $r,\ s'$
    - Choose either UPDATE(A) or UPDATE(B)
    - If UPDATE(A):
      - Define $a^* = arg\ max_a Q^A\,(s', a)$
      - $Q^A(s, a) \leftarrow\ Q^A(s, a)\ + \alpha\ (s, a)\left(r + \gamma\ Q^B\,(s', a^*) - Q^A(s, a)\right)$
    - If UPDATE(B)
      - Define $b^* = arg\ max_a Q^B\,(s', a)$
      - $Q^B(s, a) \leftarrow\ Q^B(s, a)\ + \alpha\ (s, a)\left(r + \gamma\ Q^A\,(s', b^*) - Q^B(s, a)\right)$
    - $s\ \leftarrow\ s'$
  - Until s is terminal

2.7 Deep Reinforcement Learning

The idea of deep neural networks being trained using backpropagation was first introduced in 1986 by D. E. Rumelhart, G. E. Hinton, and R. J. Williams [8]. The deep architecture utilized hidden layers to successfully learn non-linear functions. Further research in convolutional neural networks led to several major breakthroughs in computer vision. Imperviousness of neural networks to the 'curse of dimensionality' and their ability to extract high level features from raw sensory data made them a potential candidate for RL techniques with sensory data. Deep-Q learning (DQN) [9] proposed the first deep learning model that learns to control policies using RL, directly from high dimensional sensory inputs. It uses convolutional neural network to predict the Q-values and outperformed more than half of the Atari 2600 games.

2.7.1 Deep-Q Learning

Instead of using tables to store and lookup the Q-values in Q-Learning algorithm (section 2.5) , Deep Q-Learning uses non-linear function approximation to approximate $Q(s, a; \theta)$. θ represents the tunable parameters for approximation. Deep neural networks are used for this purpose. Following loss function (using Bellmans equation) is used in addition to calculate the loss in gradient descent [10]:

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

The neural network takes the state and the action as input and gives the q-values of each state. Q-table for DQN agent is updated using the following steps [3]:

1. Feedforward pass for y the current state $s \rightarrow$ predicted Q - values for each state
2. Feedforward pass for the next state $s' \rightarrow$ compute maximum of network outputs $max_{a'} Q(s',\ a')$
3. Set Q-value target for action to $r + \gamma\ max_{a'} Q(s',\ a')$. For all other actions make output equal to zero.
4. Update the weights of the network using back propagation

Experience Replay

Concept of experience replay was introduced by V. Mnih et al [9] to resolve the issue of correlated data. It is a memory module that stores the memories / experiences (tuples of state, action, reward and next state) from already explored state-space. Random experiences from this buffer are sampled at frequent epochs and used as training input for the DQN agent. Use of experience replay buffer increases the training stability. Using an experience replay buffer helps in convergence of the networks and helps to de-correlate the experiences.

Target Network

DQN traditionally uses the same neural network for estimating the target Q-values and predicted Q-values. Target Q-values shift with each iteration causing instability in estimating the Q-values and minimizing loss. Using two separate neural networks to stabilize the training was proposed as an enhancement to original DQN algorithm by V. Mnih et al [10]. Q-network is used to estimate the predicted Q-values while target-network is used to estimate the target Q-values. The weights of the q-network are updated at each timestep while the weights of the target-network are fixed and updated periodically with the weights of the q-network.

2.8 Challenges in RL

Following are the major challenges in the field RL [11]:

- The agent only receives the reward signal to converge to the optimal policy.
- The experiences of the RL agent are directly impacted by its actions and thereby leading to temporal correlations.
- he outcome of any particular action might be gathered after a many time steps. This is called as credit assignment problem [12].
- Balancing exploration and exploitation.

Chapter 3

Literature Review

Some of the earlier successes of RL include using a form of policy gradient RL to train a quadruped robot to fast walk [13], designing a controller for autonomous inverted flight on helicopter[14] and NJFun [15] – a spoken dialogue system that can provide the information about interesting things to do in New Jersey. Although these approaches were fairly successful, scaling them was difficult in high dimensions. This is due to three essential challenges in RL algorithms – space complexity (measurement of memory required for algorithm implementation), computational complexity (amount of operations that the algorithm executes in a single timestep) and sample complexity (amount of experiences / training the algorithms needs to behave optimally) [16]. In recent years, advent of deep learning has made it possible to scale RL to problems that were challenging before. S. Levine et al [17] used an RL approach with a CNN in robotics to improve hand-eye coordination. The system was used to predict the probability of gripper to result in successful grasps. DRL research also include agents that can meta – learn called deep meta-reinforcement learning [18], using Recurrent Neural Network (RNN) to store computation of "fast" RL algorithm and learning RNN's weights using a general purpose "slow" RL algorithm [19]. RNN has also been used with RL to deal with partially observable MDPs – POMDPs [20] (environment where the agent has incomplete or noisy information about the state) using policy gradient method having capability to memorize the events in the past [21, 22]. Research in DRL ranges wide from target driven visual indoor navigation [23] to learning paying video games [10]. The motivation behind learning to play video games using DRL is

to generate systems that can adapt to the challenges in the real world [11].

Function approximation and feature extraction from high dimensional sensory data using CNNs progressively resulted in their use in RL algorithms for video games. Going further, the Deep Q-Learning algorithm [9] and some of the advancements in it are reviewed in detail. Deep Q-Learning (DQN), a first deep learning model that successfully learns the control policies from raw high dimensional sensory input using RL was proposed by V. Mnih et al [9]. The algorithm was tested on Atari 2600 games [24] and resulted in a state-of-the-art benchmark for RL methods in Atari games. This variant of Q-Learning [6] algorithm accepts raw RGB screenshots of the game and uses Convolutional Neural Network (CNN) to detect objects from the high dimensional data. Hence, it resolves the issue of high dimensional observation spaces. This algorithm uses a policy network (a CNN) for the approximation of both Q-value function and optimal Q-value function. Loss or TD Error between the current Q-value function and optimal (or target) Q-value function is computed by taking the mean squared error between the two. The aim of the network is to minimize the TD error which is achieved by performing Gradient Descent. Along with this, an experience replay mechanism that samples previous transitions is also used to reduce the issue of correlated data and non-stationary distributions. Later, V. Mnih et al [10] introduced the addition of target network, that has same initial weights as the policy network. This approach uses two different neural networks to calculate target Q values (target network) and current Q values(policy network). The weights of the target network are updated with that of the policy network after a fixed number of steps. This enabled the agent to calculate the TD error between the target Q values and estimated Q values more efficiently. T. P. Lillicrap et al [25] introduced Deep DPG (DDPG) utilising experience replay, target network and deep function approximators to learn policies in continuous and high dimensional action spaces. The algorithms reduced the timesteps required for training by a factor of 20. Unlike previous advancements in DQN, authors changed the structure of neural network in [26] rather than the algorithm itself. This neural architecture decouples the value and advantage function streams in deep Q-networks. The lower layers of the network are kept convolutional. This is followed by two sequences of fully connected layers. These streams provide separate estimates of value and advantage functions, which then combine to produce a single Q function. This steered to more frequent updating of the value stream and ultimately to better ap-

proximation of value streams. A method called Human Checkpoint Replay [27] learns to play the most difficult games in Atari using checkpoints generated from human experience. These checkpoints are sampled from the gameplay of human player and are used as the starting points in the game. M. Hessel et al [28] integrated the existing improvements in DQN [9] into a single algorithm and achieved competitive results. F. Moreno-Vera [29] presented recurrent double Q-Learning as an improvement over double Q-Learning. However, the proposed method only achieved the state-of-the-art results on a few gaming scenarios.

3.1 History of Reinforcement Learning in Gaming

RL in video games is a recognized area of research and is frequently used to evaluate the performance of RL algorithms. This section presents the review of some of many significant achievements in this area.

3.1.1 TD – Gammon

In 1992 G. Tesauro proposed TD-Gammon at IBM's Thomas J. Watson Research Center [30]. It is a neural network that learns backgammon from playing against itself. It is based on a temporal difference learning algorithm called TD-Lambda, proposed by R. S. Sutton and A. G. Barto [12]. TD-Gammon algorithm uses a three-layer ANN to estimates the reward for all the possible moves (action) in any particular state. It selects the move that gives the highest reward and keeps updating the ANN parameters that are selected at random in the beginning of the game. The network learns to play at a "strong intermediate level" with minimal knowledge of just board state i.e. the network is not provided any information on how to play good backgammon. Adding few hand-crafted features to the network's input representation resulted in master level performance. The algorithm was tested numerous times in play against several good human players including former world champions Bill Robertie and Paul Magriel. The performance of TD-Gammon was very close to that of best human players (Bill Robertie managed to win the game

3.1.2 AlphaGo

D. Silver et al [31] presented AlphaGo which is a combination of deep neural networks and Monte Carlo Tree search (MCTS) [32]. It became the first algorithm to beat a professional Go player. It uses two separate neural networks to estimate value (expected future reward from any particular state in a perfect play) and policy (mapping of best actions from any particular state / position at the board). State of the game board is passed as its 19 X 19 image to CNN. First, the training of the network is done in 3 stages. A policy network is trained using supervised learning using the expert human gameplay as dataset. Second, another policy network is trained using RL that improves the previous policy network. Third, a value network is trained that predicts the winner out of all games that are played by the previously trained policy network Both these networks are integrated via MCTS. In the later versions, AlphaGo uses only RL to learn the optimal policies and achieve super human performance [33].

3.1.3 DOTA 2

OpenAI invented an AI that could beat professional players in this game, in 1v1 matches. The agent used RL techniques and learned the game completely from self-play (learning from playing with itself) [34].

Chapter 4

Flappy Bird Game

Flappy bird is a popular arcade-style game developed by Vietnamese video game artist and programmer Dong Nguyen in May 2013. The player controls the bird which moves to the right and enables it to keep flying by flapping (by tapping on the screen or pressing the up button) its wings. The task is to prevent the bird from crashing into the columns of pipes. Every time the bird passes a pipe (obstacle), player is awarded one point. At the end of the game (when the bird collides with the pipes), total score of the player is displayed (i.e. total number of pipes crossed). 4.1 shows the graphical user interface of this game.

Flappy Bird game has become a very common testing platform for RL algorithms since its first usage in [35]. This game environment is used as a testbed in this project and simulated using Pygame [36]. Pygame is a python wrapper for SDL (Simple Direct Media Layer) library which allows the users to create fully featured video games using Python.

Visual representation of the game in the dissertation is kept almost similar to the original game. The bird (enabled by the RL agent) moves towards right i.e. horizontally with constant velocity. The velocity of the bird along vertical axis is controlled by the player – which in this case is the agent. The vertical velocity of the bird keeps on increasing linearly in the downward direction until the player interferes and chooses to "flap" the bird in upward direction i.e. increase the velocity in upward direction. The game randomly generates two vertical pipes with a gap in between which are considered as obstacles for the agent to pass. The length of the upper and lower pipes

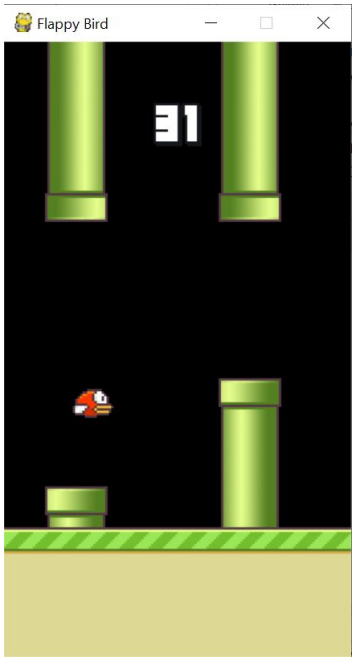

Figure 4.1: Flappy Bird game - Graphical user interface

varies randomly while maintaining the defined gap between them. The agent takes an action at every timestep which represent one iteration including Q-values calculation and selecting the best possible action.

4.1 Difficulty Levels

In this project, four models were trained with varying difficulty levels based on the vertical gaps between upper and lower pipes. The gap size between the upper and lower pipes was used to introduce significant variations for the different models and was chosen. Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5 illustrate the graphical user interface of the four difficulty models.

The models are as follows:

Low Difficulty Model – The model with the lowest difficulty has the largest vertical distance (i.e. 230 pixels) between the upper and lower pipes. The agent can learn

to perform well in this environment by learning to stay in the middle of the screen.

Moderate Difficulty Model – The distance between all the upper and lower pipes of this environment lies between that of low difficulty model and high difficulty model (i.e. 160 pixels). The agent needs to learn better than the low difficulty level as just staying in the middle of the screen all the time will not lead to very high scores.

Hard Difficulty Model – The level of difficulty for this model is very high as the distance between all the upper and lower pipes is very less (i.e. 100 pixels). This means that staying in the middle will almost always lead to very low scores and the agent has to learn not to crash while trying to pass through very small gaps.

General Model – This model generates each pipe with a random gap size (selected from: 100, 130, 160, 190, 220, 230 pixels) between the upper and lower pipes. This model is a combination of low, moderate and hard difficulty models in a way. This model better represents the obstacles or task that an AI might need to perform in real time. General model can be considered the hardest model as the gap size between upper and lower pipes is not constant. The agent needs to manage this variation and learn how to pass these obstacles.

4.2 Train and Test environments

The agent is trained on the training environment where two consecutive pairs of pipes are generated in each frame, each pair separated from the other by a constant horizontal distance. The trained models are then evaluated on both training and testing environments. In testing environment, only one pair of pipes are generated in each frame i.e. the horizontal distance between the pipes is larger than that in the training environment. Both the environments have four difficulty levels as described in section 4.1. Figure 4.6 and Figure 4.7 present the graphical user interface of training and testing environments of hard model respectively.

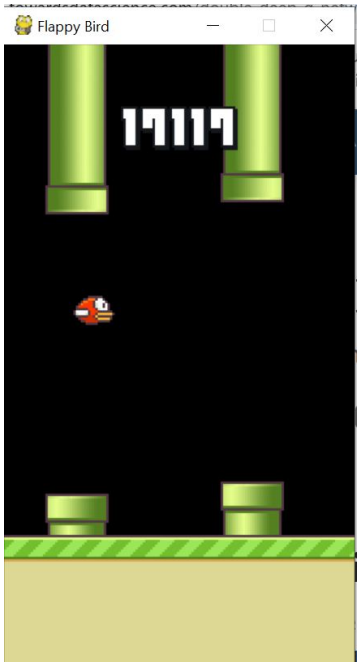

Figure 4.2: Easy model

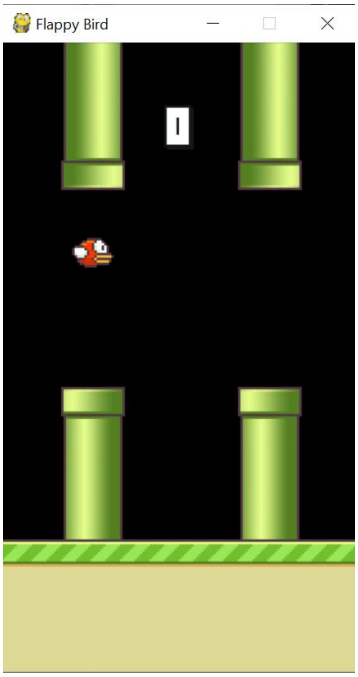

Figure 4.3: Moderate model

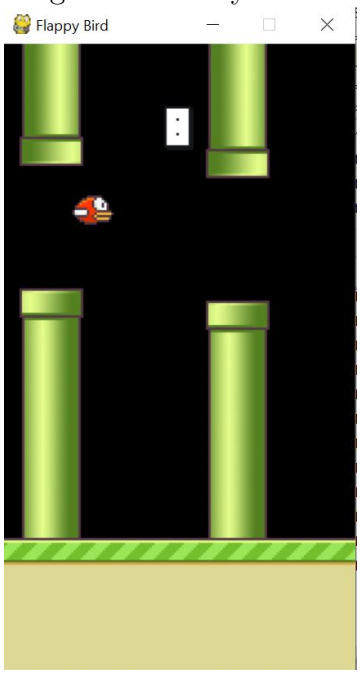

Figure 4.4: Moderate model

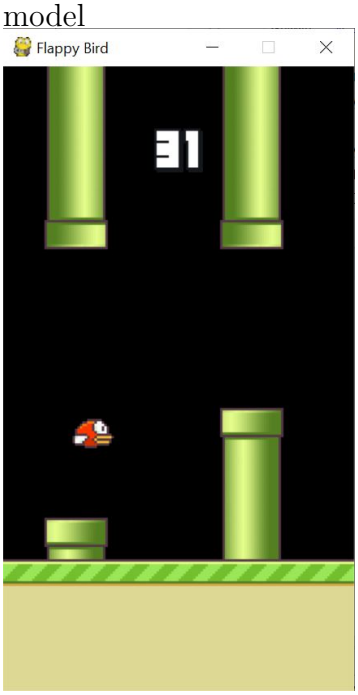

Figure 4.5: Moderate model

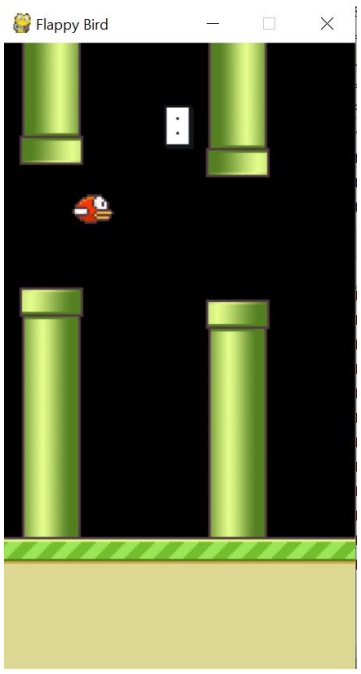

Figure 4.6: Training environment - Hard model

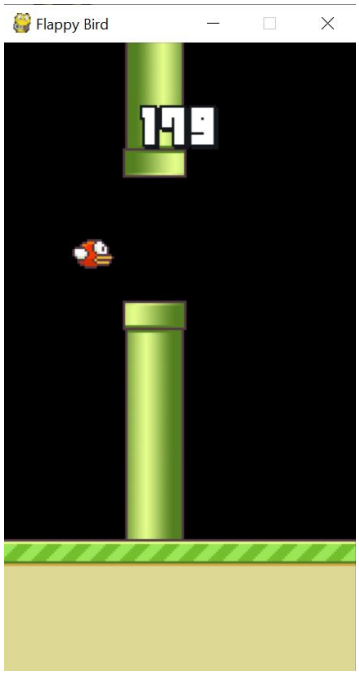

Figure 4.7: Testing environment - Hard model

4.3 Action Space

The action set for the environments contains two elements representing two actions that are same and valid for all the states including:

- Flap
- Do not flap

4.4 State Space

The state of the agent at any particular time is represented by a sequence of the frames of the game along with the sequence of the actions taken by the agent. The implementation in this project uses four consecutive frames to represent the state. This is done in order to provide the necessary temporal information for the agent to continue playing. Only with the help of multiple frames the agent determines the state in which the birds end up. First frame gives the visual information of the current state

of the bird and the last tells the next state of the words. So, the information of the current state, action taken and the resulting state constitutes as a single state for the algorithm.

4.5 Goals and Rewards

The goal of the game is to pass maximum number of pipes without colliding with one or touching the ground and the ceiling. The agent achieves this by receiving the numerical reward of +0.1 for every neutral action the keeps the agent alive, +1 for passing a pipe and -1 when the collision happens and the game ends. The ultimate aim of the agent is to increase this numerical reward over multiple games.

4.6 Implementation Details

4.6.1 Input Pre – processing

Since the frames captured from the game is high dimensional, they are pre-processed to reduce their dimensionality and eventually state space. The pixels of the frames are pre-processed which originally are in RGB format. These RGB images are converted into grayscale images and thereby reducing dimensions from three to two. These images are then resized to 80 X 80 pixels. Dimensions of these images are further reduced by normalizing them from [0, 255] to [0, 1]. The resulting images are then stacked together to form a single state for the algorithm.

4.6.2 Convolutional Neural Network Architecture

Both DQN agent (section 5.1) and DDQN agent (section 5.3) use similar structure. They use convolutional neural networks to approximate the predicted and target Q-values. Pre-processed images of latest four frames (current state) the neural networks as input and Q-values of all the states are returned. The RL agent uses this information to predict the optimal action and the resulting state and reward.

The neural network implementation uses three convolutional layers and one fully connected layer. Max pooling is used once after first convolutional layer. The architec-

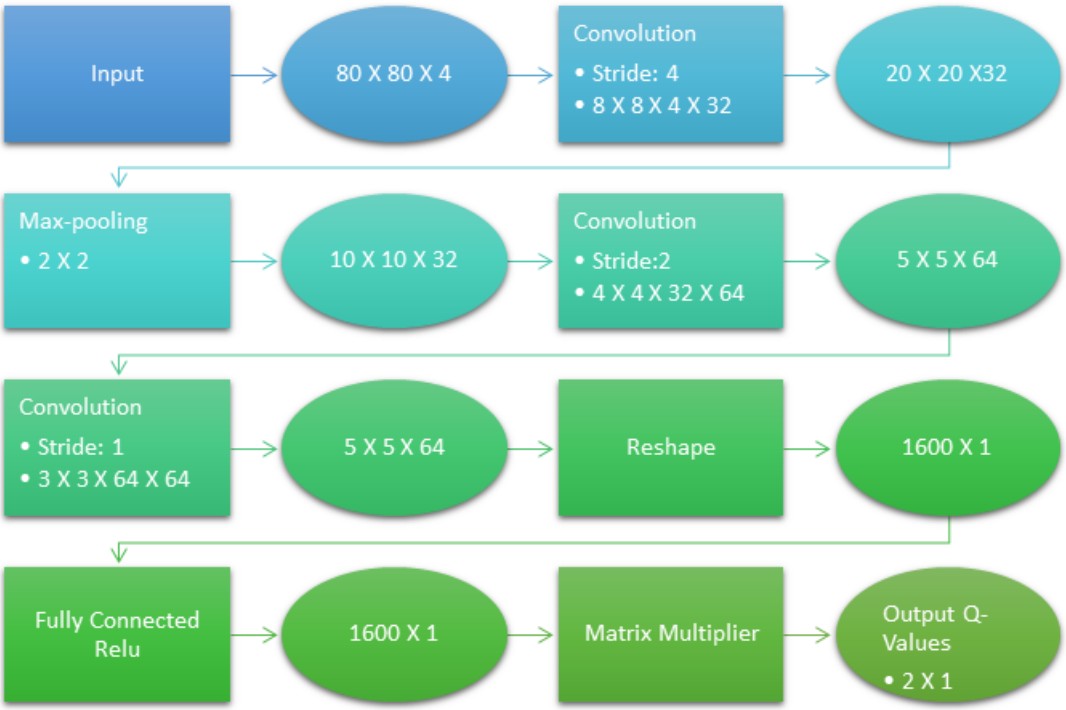

Figure 4.8: CNN architecture used in DRL algorithms

ture used for approximating Q-values in DQN and DDQN implementation is depicted in Figure 4.8.

4.6.3 Software and Hardware

The specifications of hardware are detailed in Table 4.1.

Hardware	**Description**
Processor	Intel® Core™ i3-8145U CPU @ 2.10 GHz 2.30 GHz
RAM	4.00 GB

Table 4.1: Hardware specifications

Python was chosen as a programming language as it helps in developing a clean, read-

able and concise code Along with it, deep learning framework Tensorflow [37] was used for neural network implementation. Many other external libraries such as OpenCV [38] (for image processing) and Numpy [39] (for operations on matrices) were used in this project. PyCharm was used as IDE and project development and versioning was managed using GitHub. Table 4.2 contains the details regarding software specifications.

Software	**Description**
Operating System	Windows 10
Programming language & version	Python 3.7.3
IDE	PyCharm 2019 .3.3 (Professional Edition)
Deep Learning Framework	Tensorflow 2.1.0
Version control	Git

Table 4.2: Software specifications

Chapter 5

Experiments

Different RL agents (experiments) are designed using various algorithms and strategies. The agents are trained until they started to give superhuman performance or when more training stopped having any effect on the agent's performance. The performance of the agent was judged using the game score (number of pipes agent passes). Figure 5.1 illustrates the three types of agents that were implemented and their corresponding exploration decay strategies.

5.1 DQN Agent – Baseline Model

Deep Q-Learning algorithm using experience replay [9] is implemented as the baseline model (DQN agent) for this project. Algorithm 3 depicts the pseudo code for the implemented DQN agent. Decaying epsilon greedy strategy (explained in section 2.4.3) is used in this implementation to balance the trade-off between explorations and exploitation. The agent records a few experiences / observations as tuples $< currentstate, action, reward, nextstate >$ in the replay memory before starting the training. A neural network (Q-network) is used to predict the Q-values which is done by minimizing the loss function using gradient descent. The DQN agent is trained on different scenarios with varying difficulty (easy, moderate, hard and general). It was observed that in the beginning of the training, the agent keeps on selecting the up or

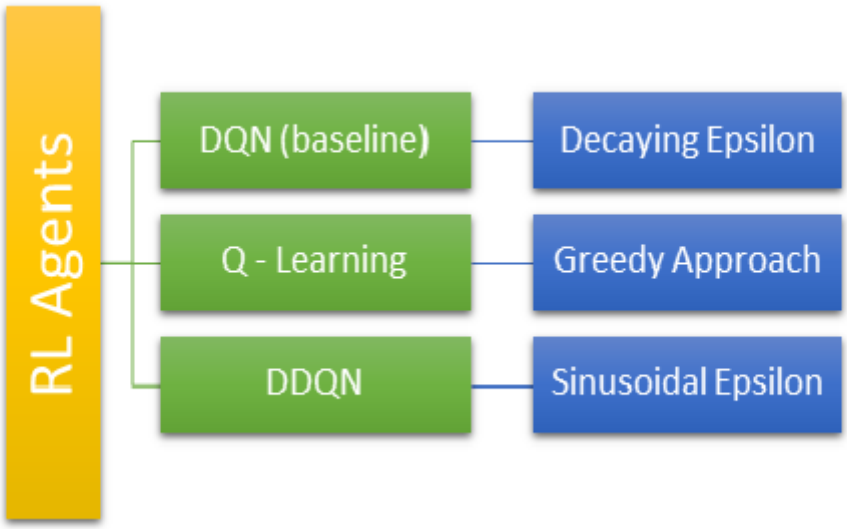

Figure 5.1: RL agents

down action until the bird reaches the ceiling / floor of the game. It takes the agent long time to learn to try and stay in the middle of the screen instead. Hence, for general model, to speed-up the training, trained easy model was fine-tuned instead of training from the scratch. Table 5.1 contains the details of the parameters used in this implementation.

Algorithm 3: DQN

- Algorithm parameters: $\gamma(gamma) \geq 0, \varepsilon_o(initialepsilon) \geq 0, \varepsilon(epsilon) \geq 0, \varepsilon_f(finalepsilon) \geq 0$
- Initialise replay memory D
- Initialise $OBSERVE, EXPLORE$
- Initialise Q-value function $Q(s,a), \forall s \in S, a \in A$ via q-network with random weights
- Observe initial state s
- $\varepsilon \leftarrow \varepsilon_o$
- Loop for each episode until convergence:
  - Select a random action a with probability ε
    otherwise
    $a \leftarrow argmax\ Q(s', a')$
  - Take action a
  - Observe reward r and next state s'
  - Store experience $< s, a, r, s' >$ in D
  - If $\varepsilon > \varepsilon_f and t(timestep) > OBSERVE$:
    $\varepsilon \leftarrow \varepsilon - \left(\frac{\varepsilon_o - \varepsilon_f}{EXPLORE}\right)$
  - If $t > OBSERVE$
    - $minibatch < ss, aa, rr, ss' > \leftarrow$ sample random experiences from D
    - Calculate output y for each experience in minibatch:
      $y \leftarrow rr$ if ss' is terminal (end state)
      otherwise
      $y \leftarrow rr + \gamma * maxQ(ss', aa')$
    - Optimize q-network using gradient descent and minimizing loss function:
      $(y - Q(ss, aa))^2$
    - $s \leftarrow s'$

Parameter	Initial Value	Description
gamma	0.99	Decay rate of past observations
initial epsilon	0.1	probability of selecting random actions in the beginning of training
final epsilon	0.0001	Probability of selecting random actions towards the end of the training
OBSERVE	10000	Number of timesteps till the agent observes and records the experiences before starting training
EXPLORE	3000000	Number of timesteps after which the agents stops exploring

Table 5.1: DQN parameters

5.2 Q-Learning Agent

A variation of Q-Learning algorithm was proposed [40](Vu and Tran 2020), which according to the author outperforms the "more complex deep Q-Learning approaches". Q-Learning agent's implementation was influenced by this approach.

Process of extracting the state information for this algorithm is different than the one used for the other agents. The algorithm receives the following information from the game environment to constitute the current state of the game.

- x_diff = The horizontal difference between the bird and the next pipe
- y_diff = The vertical distance between the bird and the bottom pipe
- y_vel = The in-game velocity of the bird

Another technique that is proposed to make the algorithm converge faster is to use backward updates. This means feeding the experiences to the agent in reverse order (i.e. last experience first) while the agent is learning from the stored experiences. As a result, the agent will get more useful information first i.e. when agent collided with the pipe and therefore learns what not to do first. In addition to this, authors used epsilon

greedy approach to balance exploitation and exploration for the agent. Algorithm 4 contains the pseudo code used by to implement Q-Learning agent.

The model used by the authors to train and evaluate the agent was similar to the hard model created in this project. The Q-Learning agent was trained on the general model in our implementation. The agent was trained using the approach presented, but performed poorly than the baseline. Since the Q-Learning agent's performance was not matching that of the baseline, the agent was trained only on the general model. Due to time crunch, a decision was made to not train this agent for easy, moderate and hard models.

Algorithm 2: Backward Q-Learning

- Algorithm parameters: $\gamma(gamma) \geq 0, \varepsilon(epsilon) \geq 0, \eta$
- Initialise all Q-values to 0
- Loop for each episode until convergence:
  - Initialise memory M
  - Observe initial state s
  - Loop until s is not terminal
    - Select a random action a with probability ε
      otherwise
      $a \leftarrow max_a \left(Q\left(s', a'\right)\right)$
    - Take action a
    - Observe reward r and next state s'
    - Observe reward r and next state s'
    - $s \leftarrow s'$
  - For observation in M do:
    $Q\left(s, a\right) \leftarrow \left(1 - \eta\right) Q\left(s, a\right) + \eta \left(r + \gamma \ maxQ\left(s', a'\right)\right)$

5.3 DDQN Agent

Issue of overestimating the action values was also observed in DQN algorithm. The idea behind Double Q-Learning [7] was utilised to develop Double Deep Q-Learning algorithm [2]. This adaptation of DQN algorithm reduces the overestimations and increases the performance. The algorithm is implemented in an attempt to beat the baseline algorithm (section 5.1). This implementation uses sinusoidal exploration decay strategy as described in section 2.4.4. Algorithm 5 contains the pseudo code for the same. The agent records a few experiences / observations as tuples $< currentstate, action, reward, nextstate >$ in the replay memory before starting the training. A neural network (Q-network) is used to approximate the Q-values while another neural network (target-network) predicts the target Q-values. The Q-network minimizes loss function using gradient descent. DDQN agent was trained on different models based on different difficulty levels – easy, moderate, hard and general. Table 5.2 contains the details of the parameters used in this implementation.

Algorithm 5: Double Deep Q-Learning with Sinusoidal exploration decay

- Algorithm parameters: $\gamma(gamma) \geq 0, \varepsilon_o(initialepsilon) \geq 0, \varepsilon(epsilon) \geq 0, \varepsilon_d(epsilondecayrate) \geq 0$
- Initialise replay memory D
- Initialise $OBSERVE, EXPLORE, TARGET\_UPDATE\_THRESHOLD$
- Initialise q-network with random weights
- Initialise target-network with q-network weights
- Observe initial state s
- $\varepsilon \leftarrow \varepsilon_o$
- Loop for each episode until convergence:
  - Select a random action a with probability ε
    Otherwise,
    $a \leftarrow argmaxQ\left(s', a'\right)$
  - Take action a
  - Observe reward r and next state s'
  - Store experience $< s, a,\ r,\ s' >$ in D
  - If $t(timestep) > OBSERVE$:
    $\varepsilon \leftarrow (\varepsilon_o * \varepsilon_d) * \left(\frac{1}{2} * \left(\frac{1+cos(2\pi t)}{EXPLORE}\right)\right)$
  - If $t > OBSERVE$
    - $minibatch < ss, aa, rr, ss' > \leftarrow$ sample random experiences from D
    - Calculate output y for each experience in $minibatch$:
      $y \leftarrow rrif ss'$ is terminal (end state)
      otherwise
      $y \leftarrow rr + \gamma * maxQ_{target}\left(ss', argmax\left(Q\left(ss',\ aa'\right)\right)\right)$
    - Optimize q-network using gradient descent and minimizing loss function:
      $\left(y - Q\left(ss, aa\right)\right)^2$
    - $s \leftarrow s'$
    - After every $TARGET\_UPDATE\_THRESHOLD$ iterations: $weights\left(target - network\right) \leftarrow weights\left(q - network\right)$

Parameter	Initial Value	Description
gamma	0.99	Decay rate of past observations
initial epsilon	0.1	probability of selecting random actions in the beginning of training
epsilon decay rate	0.99	Probability of selecting random actions towards the end of the training
OBSERVE	10000	Number of timesteps till the agent observes and records the experiences before starting training
EXPLORE	3000000	Number of timesteps, used to compute epsilon
TARGET UPDATE THRESHOLD	1000	Weights of target network are updated after every 1000 iterations

Table 5.2: DDQN parameters

Chapter 6

Results and Analysis

This project aims at comparing the performance of reinforcement learning algorithms using the popular game Flappy Bird as a test-bed. The performance of the trained models is compared using metrics: game score (i.e. number of pipes passed by the agent) – average score and maximum score and training time.

Three different agents were trained and evaluated – DQN (baseline), Q-Learning and DDQN. Due to the poor performance of the Q-Learning agent (section 5.2) on general difficulty model, I did not continue the training for the rest of the scenarios and it is not considered as part of the comparative study.

Table 6.1 and Table 6.2 present the performance of DQN agent and DDQN agent respectively evaluated in the training environment. Table 6.3 and Table 6.4 on the other hand contain the results of the performance of the two agents evaluated on testing environment.

Performance of the easy model for DQN agent in training environment was evaluated only over two games. The agent had reached more than 1000000 timesteps and therefore was stopped after two games due to time constraints. Due to the same reason, performance of DQN agent for moderate model, DDQN agent for easy model and DDQN agent for moderate model in training environment were evaluated over 12, 1 and 10 games.

DQN agent for general model was finetuned on the trained DQN easy model. Hence, the total number of training iterations for general model is the sum of iterations needed to train easy model and the additional iterations DQN agent took to finetune over gen-

eral model (2950000 + 10190000).

	Easy Model	**Moderate Model**	**Hard Model**	**General Model**
Maximum Score	18720	10092	204	503
Average Score	14530.5	4567.33	67.8	93.18
Training Iterations	2950000	4320000	9220000	2950000 + 10190000 = 13140000
Total games / episodes	2	12	100	100

Table 6.1: DQN agent results - training environment

	Easy Model	**Moderate Model**	**Hard Model**	**General Model**
Maximum Score	28022	13136	1878	2396
Average Score	28022	5275.5	450.42	1023.16
Training Iterations	1690000	1470000	7740000	7520000
Total games / episodes	1	10	100	100

Table 6.2: DDQN agent results - training environment

The evaluation of the DQN and DDQN agents in the testing environment was ran for either 50 games or a maximum of 2100000 timesteps, whichever the agents reached

first.

	Easy Model	**Moderate Model**	**Hard Model**	**General Model**
Maximum Score	427	2850	268	418
Average Score	105.3	1130.83	125.52	109.5
Total games / episodes	50	24	50	50

Table 6.3: DQN agent results - testing environment

	Easy Model	**Moderate Model**	**Hard Model**	**General Model**
Maximum Score	28378	9311	1796	553
Average Score	28378	746.08	770.36	168.66
Total games / episodes	1	38	50	50

Table 6.4: DDQN agent results - testing environment

Performance of the two DRL agents – DQN and DDQN were compared across four models (easy, moderate, hard and general) in two environments (train and test). Following metrics were used for the comparative study:

- *Training Iterations* – Total number of iterations (timestep) it took the agent for training.
- *Maximum Score* – Maximum game score achieved by the agent among total games it was tested for. It is the best agent has performed.

- *Average Score* – Mean of the game score achieved by the agent across total number of games it is tested for. It describes the performance of the agent better in terms of consistency.

Figure 6.1, Figure 6.2, Figure 6.3 and Figure 6.4 shows the plot of maximum score and average score metrics achieved by the two agents for easy, moderate, hard and general models respectively. The DDQN agent outperform the DQN agent across most models and environments significantly. Moderate and general models in testing environment are an exception and almost matches the DDQN agent's performance . In addition, DDQN agent takes significantly less time to train in all the models when compared with DQN agent. The comparison is shown in Figure 6.5.

6.1 Sinusoidal Exploration Decay vs Epsilon Greedy Decay

In this project, DQN agent uses decaying epsilon greedy strategy to handle exploration – exploitation dilemma while DDQN agent uses sinusoidal exploration decay strategy. Figure 6.6 represent the plot of decaying epsilon values with respect to the training iterations for the DQN agent while training the general model. Figure 6.7 and Figure 6.9 shows the changing scores as the training progress for DQN and DDQN agents respectively. Figure 6.8 shows the changing values of epsilon in sinusoidal manner while training the DDQN agent for general model. Large values of epsilon caused the agent to die frequently and therefore leading to low scores while low values resulted in higher scores while training. This can be concluded from the plot in Figure 6.7 and Figure 6.9.

Figure 6.10 is a comparison plot of scores with respect to the training iterations for DQN and DDQN agent for general model. The plot shows that the DDQN agent starts to converge faster when compared to the DQN agent. Also, the training of the DQN agent becomes flat after epsilon decays completely, but the training of DDQN agent continues in sinusoidal fashion and reached the peak thrice. This means that the DDQN agent does not stop exploring altogether after reaching its local minima. DQN

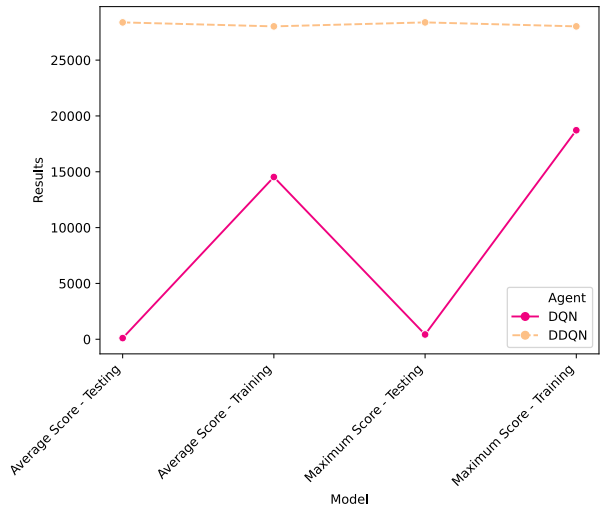

Figure 6.1: Performance plot - Easy model

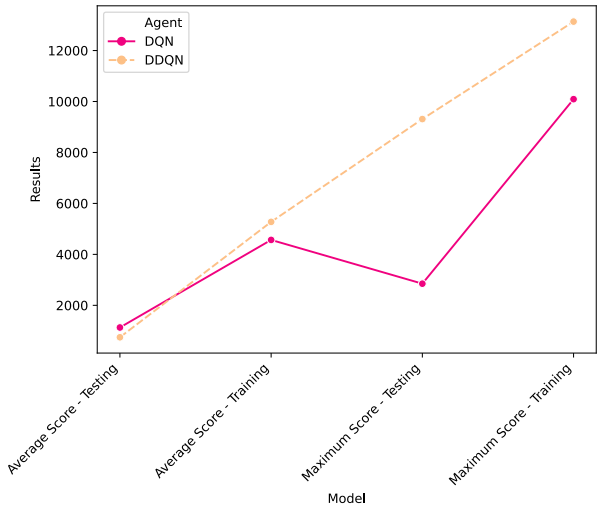

Figure 6.2: Performance plot - Moderate model

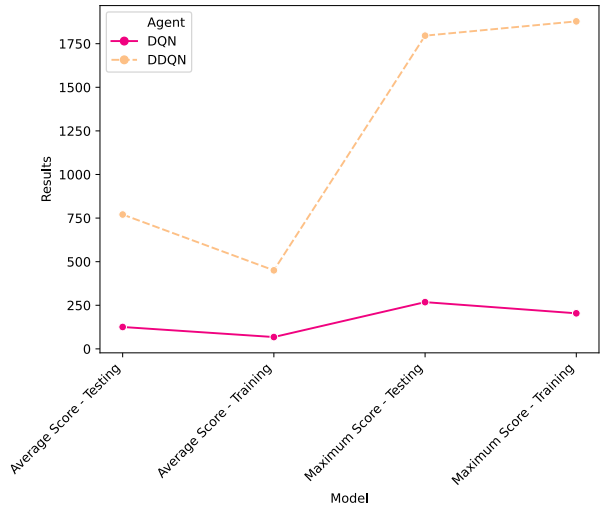

Figure 6.3: Performance plot - Moderate model

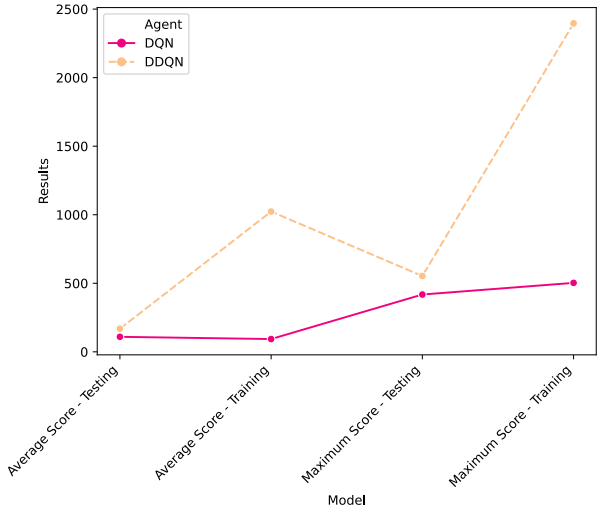

Figure 6.4: Performance plot - Moderate model

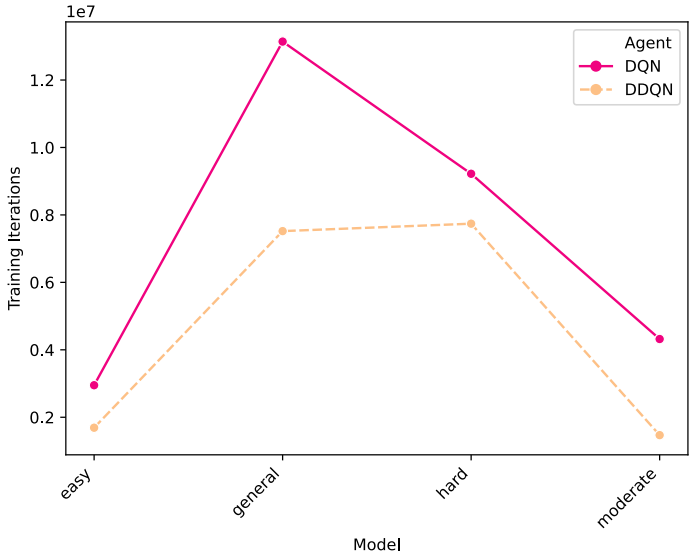

Figure 6.5: Training Iterations plot

agent on the other hand stops exploration and consequently further training once the epsilon decays completely.

6.2 Discussion

It is clear from the results of the experiments that DDQN algorithm with sinusoidal exploration decay outperforms DQN algorithm with decaying greedy epsilon substantially. Also, DDQN agent converged considerably faster. During the training of the

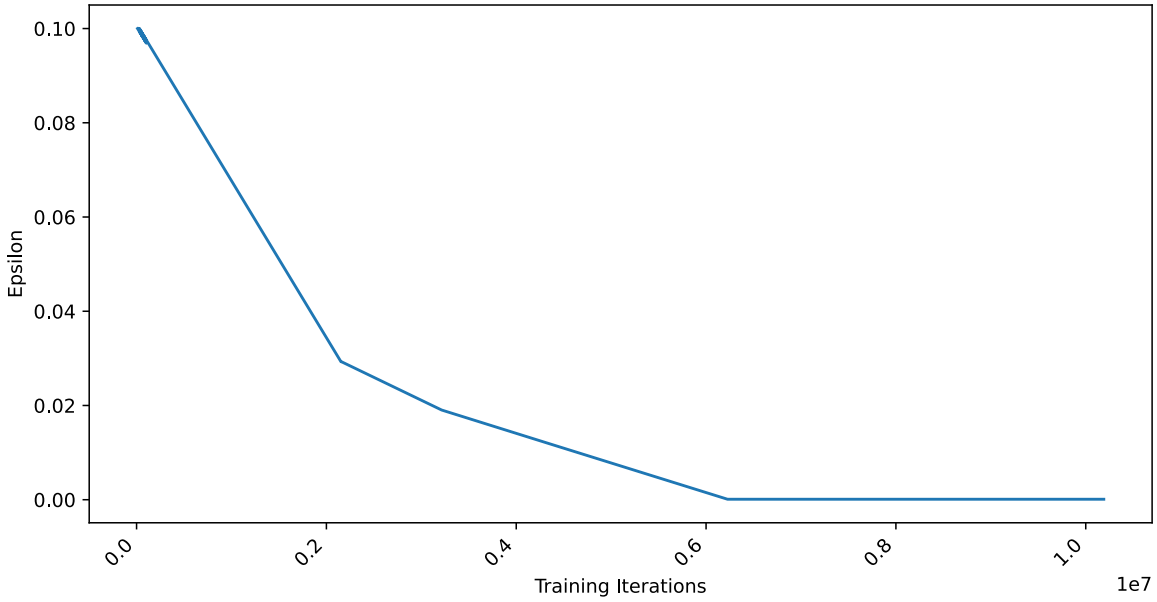

Figure 6.6: Epsilon vs Training Iterations plot for DQN agent - General model

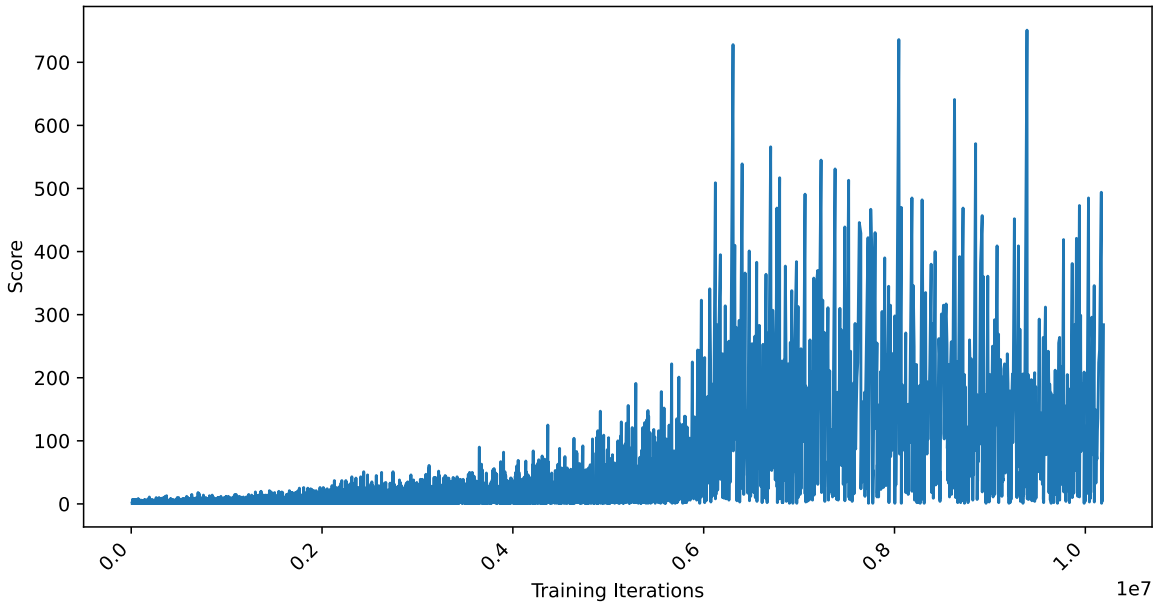

Figure 6.7: DQN agent's Score vs Training Iterations plot - General model

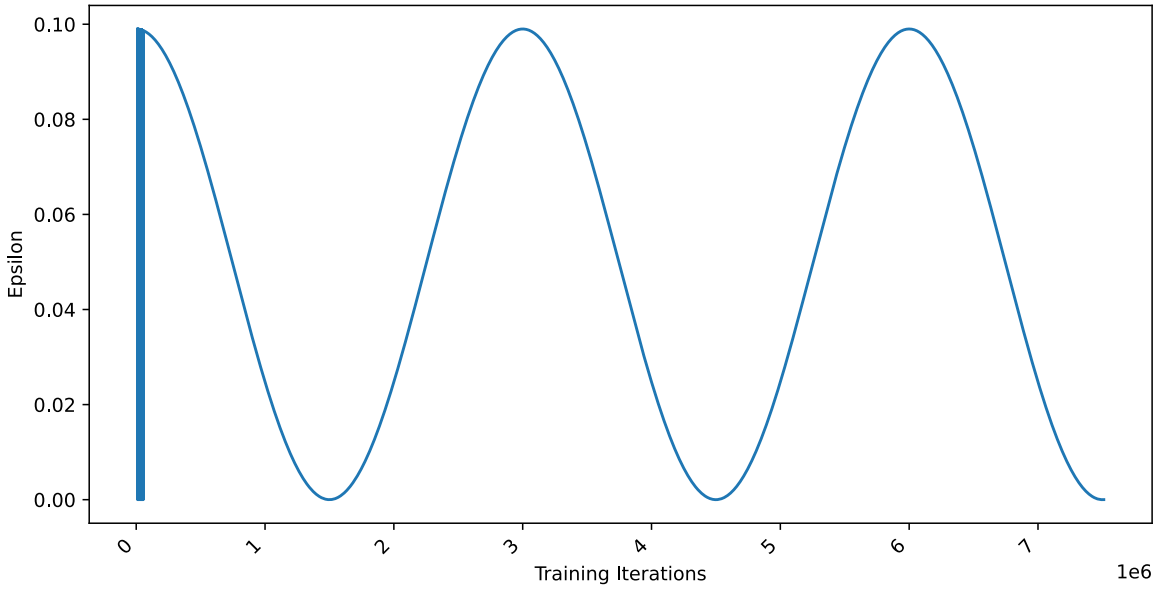

Figure 6.8: Epsilon vs Training Iterations plot for DQN agent - General model

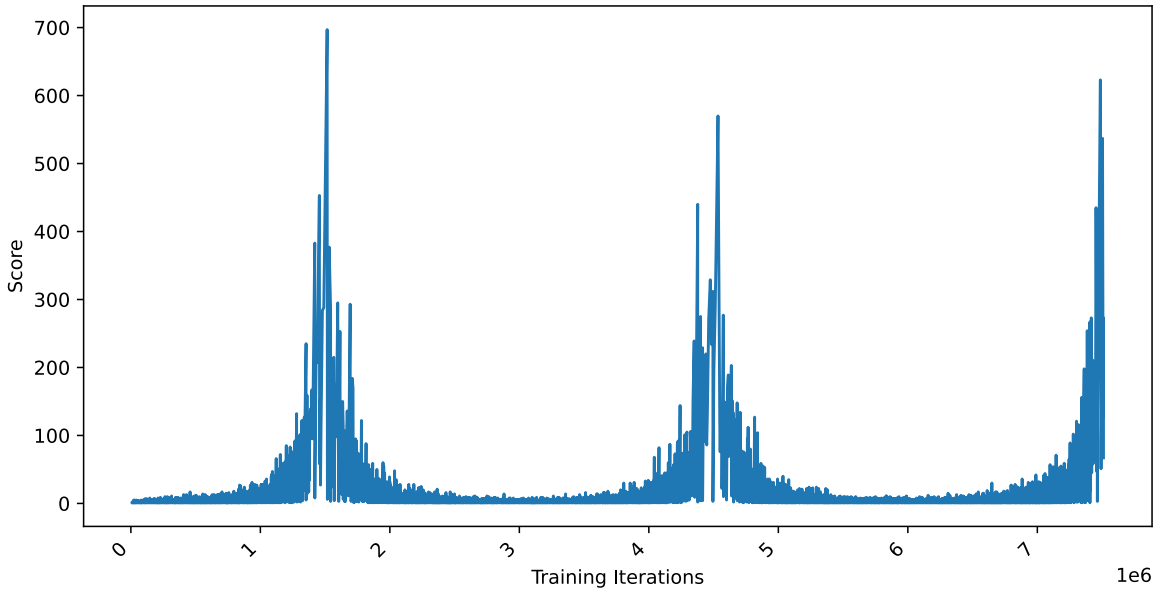

Figure 6.9: DDQN agent's Score vs Training Iterations plot - General model

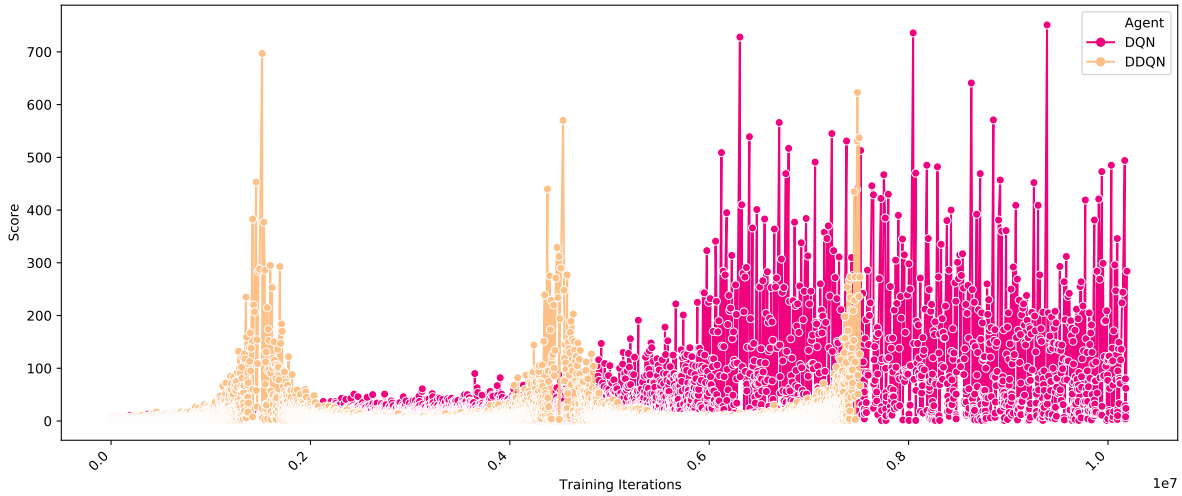

Figure 6.10: Score vs Training Iterations plot - comparison

agents, it was observed that with the level of difficulty in the models (easy, moderate, hard and general), training time of the agent also increased. Hence, not only the training times of the easy and moderate models are very low comparatively, they also achieved the superhuman score of more than 10000 (and 28000 for DDQN easy model) in the training environment. Training hard and general models on the other hand were time intensive and their performance do not reach anywhere near the easy and moderate models. DQN agent found it difficult to converge for the hard and general models and did not perform well comparatively. DDQN agent's performance and convergence even for hard and general models was very good. Even though it did not match the scores of easy and moderate models, it reached the maximum scores of more than 2300 and average score of 1000 for general model. Since hard and general models are better representations of real-time obstacles, outperforming baseline in these constitute good results.

Apart from using the target network and double deep Q-Learning algorithm for DDQN agent, using sinusoidal exploration decay function also helped. While training DQN agent, the agent stopped exploring altogether after it reached the final epsilon value. As a result, the further training of the agent stopped (or slowed down sizeably) after reaching a certain point. Sinusoidal exploration decay function in DDQN agent prevents the agent to stop exploring. Every time epsilon reaches its smallest vale, it starts increasing again and consequently agents starts to exploring more.

The performance of the agent was evaluated on two environments : train (which is the same environment agents are trained on) and test (slightly different than training environment). The agent's performance on the testing environment are generally poor than their performance in the training environment. This may be due to the fact that the algorithms use the frame representation of the games to store the state information. When we change the environment (GUI) of the game even slightly, for the algorithms it becomes an entirely new state which has not been explored yet. As a result, the agents make its best approximation and tries to achieve the maximum score. Fine-tuning the agent by training it sometime in the new environment might lead to better results.

It has also been observed, that the agents might start to forget or begin to re-learn the policies if trained too much.

Chapter 7

Conclusion

In this thesis we analysed the performance of the RL agents trained in a classic arcade gaming environment using DDQN and DQN techniques. Different epsilon decay strategies are utilised in order to identify their characteristics and effects on agents' performance. Baseline model (DDQN agent) was first created followed by other models. Two different strategies to balance exploration and exploitation were also studied and used in both the agents. The details of the experiments are outlined in section 5. Different game environments (test and train) and models (easy, moderate, hard and general) were also designed to support evaluation of the algorithms and strategies used (section 4).

Three performance metrics were defined to evaluate the performance of RL agents: maximum game score, average game score and the number of iterations agent took to train on different models. These metrics were computed for the agents and presented. The performance of the agents was then compared with the each other using the three defined metrics. The agents were trained and evaluated on different models, to see the impact of "difficulty" on the ease of training and performance. The agents were also evaluated on two different environments to measure their flexibility.

The findings of these experiments indicate that the sinusoidal decay strategy can significantly improve the performance of the AI agent while exposed to unknown variation in the environment which in this thesis is simulated by introducing difficulty levels. It was also observed that the DDQN network architecture results in a significantly less training time and using the target-network during the training stabilised the Q-values

estimation.

Importantly, we note that the main challenge in such platform and systems are:

- Balancing exploration and exploitation which is one of the most demanding challenge of all. A single strategy may not work for all the environments and may vary from environment to environment.
- In more complicated environments, the DRL agent might get stuck in the local optimum. Simple exploration decaying strategies might not be enough to escape local optima and reach global optimum in such environments.
- Complete training of RL agents from the scratch takes a long time (weeks for complicated environments). Even then, it might not be possible for the agent to explore all the scenarios.
- It was observed that if the agent is over trained, it starts to perform worse i.e. forgets what it has learned before. This leads to the need of knowing the correct time when the training of the agent should be stopped.

7.1 Future Work

There are numerous possibilities to extend the work done in this project. An obvious way to extend this study could be to implement more algorithms and strategies for the comparative study. Another area that I wanted to explore but could not due the time constraint was to implement a DDQN algorithm using decaying epsilon greedy strategy. This would have enabled me to bifurcate the impact of using double Q-Learning algorithm and sinusoidal exploration decay strategy on the performance. It was observed that after convergence, DRL agents started to perform poorly if kept on training. It could be a good idea to study the cause for this and research on techniques to prevent this.

Furthermore, more experiments can be conducted for better evaluation of the agent's performance. Different game environment or models could be used for another comparative study.

7.1.1 Offline Learning in DRL

R. Agarwal, D. Schuurmans, and M. Norouzi [41] proposed an approach to train the DQN agent [9] using offline learning. DQN agent was trained using logged replay data of previously trained DQN agents instead of learning online from the environment. The authors also contributed a large dataset consisting 50 million logged experiences (tuple of observation, action, reward, next observation) of DQN agents for more than 60 Atari games. Since in online learning, agent needs to gather a very large set the experiences from scratch, training is a long process. Also, it is not possible for a single RL agent to explore all the scenarios in a constraint time frame. Offline learning is effective and can overcome this barrier if a "sufficiently large and diverse dataset is available".

Utilizing this approach to enhance the performance of the DDQN agent further was considered for this dissertation. Unfortunately, the replay logs shared by [41] did not contain the dataset for Flappy Bird environment. For future research, a diverse and large dataset containing the replay logs of DQN agent in Flappy Bird environment can be accumulated. This dataset can then be used for offline training of the RL agent in the current implementation and its evaluation.

Bibliography

[1] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[2] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *arXiv preprint arXiv:1509.06461*, 2015.

[3] R. Chuchro and D. Gupta, "Game playing with deep q-learning using openai gym," *Semantic Scholar*, 2017.

[4] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," 2011.

[5] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[6] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[7] H. V. Hasselt, "Double q-learning," in *Advances in neural information processing systems*, pp. 2613–2621, 2010.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

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

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[11] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[12] R. S. Sutton, A. G. Barto, *et al.*, *Introduction to reinforcement learning*, vol. 135. MIT press Cambridge, 1998.

[13] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, vol. 3, pp. 2619–2624, IEEE, 2004.

[14] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental robotics IX*, pp. 363–372, Springer, 2006.

[15] S. Singh, D. Litman, M. Kearns, and M. Walker, "Optimizing dialogue management with reinforcement learning: Experiments with the njfun system," *Journal of Artificial Intelligence Research*, vol. 16, pp. 105–133, 2002.

[16] A. L. Strehl, L. Li, E. Wiewiora, J. Langford, and M. L. Littman, "Pac model-free reinforcement learning," in *Proceedings of the 23rd international conference on Machine learning*, pp. 881–888, 2006.

[17] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[18] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *arXiv preprint arXiv:1611.05763*, 2016.

[19] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "Rl ²: Fast reinforcement learning via slow reinforcement learning," *arXiv preprint arXiv:1611.02779*, 2016.

[20] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[21] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Recurrent policy gradients," *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 620–634, 2010.

[22] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, "Memory-based control with recurrent neural networks," *arXiv preprint arXiv:1512.04455*, 2015.

[23] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *2017 IEEE international conference on robotics and automation (ICRA)*, pp. 3357–3364, IEEE, 2017.

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

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

[26] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, pp. 1995–2003, 2016.

[27] I.-A. Hosu and T. Rebedea, "Playing atari games with deep reinforcement learning and human checkpoint replay," *arXiv preprint arXiv:1607.05077*, 2016.

[28] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *arXiv preprint arXiv:1710.02298*, 2017.

[29] F. Moreno-Vera, "Performing deep recurrent double q-learning for atari games," in *2019 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, pp. 1–4, IEEE, 2019.

[30] G. Tesauro, "Td-gammon, a self-teaching backgammon program, achieves master-level play," *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

[31] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[32] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*, pp. 72–83, Springer, 2006.

[33] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[34] OpenAI, "Openai five." `https://blog.openai.com/openai-five/`, 2018.

[35] K. Chen, "Deep reinforcement learning for flappy bird," 2015.

[36] P. Shinners, "Pygame." `http://pygame.org/`, 2011.

[37] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[38] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[39] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[40] T. Vu and L. Tran, "Flapai bird: Training an agent to play flappy bird using reinforcement learning techniques," *arXiv preprint arXiv:2003.09579*, 2020.

[41] R. Agarwal, D. Schuurmans, and M. Norouzi, "An optimistic perspective on offline reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2020.