Performance Profiles of Basic, Memory-Based and Hybrid Deep Reinforcement Learning agents with B-Suite

Master Thesis

Author:
Rohan Das

Examiner and Supervisor:
Prof. Dr. rer. nat. habil. Gunter Saake

2nd Examiner:
Dr.-Ing. Christoph Steup

Supervisor:
MSc. Gabriel Campero Durand

Magdeburg, 08.10.2021
Rohan Das

Performance Profiles of Basic, Memory-Based and Hybrid Deep Reinforcement Learning agents with B-Suite

Abstract

In recent times, deep reinforcement learning (DRL) has emerged as a powerful approach to solve sequential decision-making tasks. On its early days, DRL was mainly restricted to controlled environments such as those from video games or robotic tasks. With the standardization of DRL, now a large amount of models proposed under this approach have been able to reach various breakthroughs, attaining superhuman level performance in complex tasks. More and more research is going into the application of DRL algorithms to solve various real-world problems. However, the evaluation of DRL algorithms has been traditionally limited to specific environments or tasks, and there is to date no standardized approach towards evaluating holistically the performance profile of DRL agents to understand how they fare in the different challenges that constitute the learning problem of a task. Hence, it is difficult for researchers to understand when facing a new problem, what DRL model or configuration thereof, to bring as a starting point.

In this research, we aim to address this problem through the experimental evaluation of a general approach towards the performance profiling of DRL agents. We base our research on Bsuite, a novel benchmarking tool proposed for the study of the core capabilities of DRL agents across tasks. Specifically, this benchmarking tool enables us to test agents at their core capabilities of Generalization, Exploration, Credit Assignment, Scale, Noise and Memory, apart from basic overall capabilities.

In our research we specifically seek to identify, with the help of B-suite, the role of models, hyperparameters and architectures of state-of-the-art DRL algorithms on the overall performance profile of the agents. Furthermore, we seek to understand if the factors contributing to certain capabilities of agents can be combined in hybrid agents that are able to present a mix of capabilities. We report results where some hyper-parameter and architectural changes are able to improve some model capabilities, leading to agents with overall better profiles. We also report some limited improvements to selected capabilities from hybrid agents.
Acknowledgements

First and foremost, I would like to express my sincere gratitude to Gabriel Campero Durand for introducing me to deep reinforcement learning, motivating me during the mid phases of this work as we faced hardships during the research, always helping me out with my doubts and providing me the necessary guidance to complete this research.

I would also like to thank Prof. Dr. Gunter Saake for making this thesis topic available at the DBSE research group.

Besides my supervisors, I would like to thank the DBSE research group, to help me out with the availability of the servers, which helped me to execute the research experiments.

Finally, I would like to thank my family, as they always motivated me during the hardships faced during this work and always supported me in the pursuit of my happiness.
Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Signature

Place, Date
# Contents

## List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3</td>
<td>19</td>
</tr>
<tr>
<td>2.3.4</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>35</td>
</tr>
<tr>
<td>4.1.1</td>
<td>35</td>
</tr>
<tr>
<td>4.1.2</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1</td>
<td>38</td>
</tr>
<tr>
<td>4.2.2</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>41</td>
</tr>
<tr>
<td>5.1.1</td>
<td>42</td>
</tr>
<tr>
<td>5.1.2</td>
<td>48</td>
</tr>
<tr>
<td>5.1.3</td>
<td>52</td>
</tr>
<tr>
<td>5.1.4</td>
<td>53</td>
</tr>
<tr>
<td>5.1.5</td>
<td>54</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Motivation ................................................. 5
1.2 Main Contributions ........................................ 7
1.3 Structure ................................................. 7

## 2 Background

2.1 Introduction to Reinforcement Learning ......................... 9
2.2 Classification of Reinforcement Learning Algorithms ........... 12
2.3 Deep Reinforcement Learning Algorithms ........................ 14
    2.3.1 DQN ................................................. 14
    2.3.2 BootDqn ............................................ 17
    2.3.3 A2C ................................................. 19
    2.3.4 R2D2 ................................................. 20
2.4 Literature Overview .......................................... 22
    2.4.1 Methodology ......................................... 22
    2.4.2 Literature Survey(DRL Bench marks) ................... 23

## 3 Design

3.1 Research Questions .......................................... 25
3.2 Design: Profiling of Agents ................................ 25
    3.2.1 Tunable Parameters .................................. 26
    3.2.2 Bsuite Environments ................................ 31

## 4 Experimental Set-Up

4.1 DRL Frameworks: ............................................ 35
    4.1.1 Bsuite: ............................................. 35
    4.1.2 Acme: ............................................... 36
4.2 Softwares, Versions and Code Configurations: ................... 38
    4.2.1 Softwares and Versions: ................................ 38
    4.2.2 Code Structure and Configurations: ................... 39

## 5 Evaluation and Results

5.1 Experiments: Research Question 1(Hyper-parameter Changes) .... 42
    5.1.1 DQN Agent ............................................ 42
    5.1.2 BootDqn Agent ....................................... 48
    5.1.3 A2C Agent ............................................ 52
    5.1.4 R2D2 Agent ........................................... 53
    5.1.5 Summary of RQ1 Exp’s: ................................ 54
5.2 Experiments: Research Question 2 (Architecture Changes) ........................................... 55
   5.2.1 DQN Agent ........................................................................................................ 55
   5.2.2 A2C Agent ......................................................................................................... 58
   5.2.3 Summary of RQ2 Exp’s: .................................................................................. 59
5.3 Experiments: Research Question 3 (Hybrid Agents) ..................................................... 60
   5.3.1 A2C Hybrid Agent .......................................................................................... 60
   5.3.2 R2D2 Hybrid Agent ....................................................................................... 62
   5.3.3 Summary of RQ3 Exp’s: .................................................................................. 64

6 Conclusion and Future Work ......................................................................................... 67
   6.1 Conclusion ............................................................................................................. 67
   6.2 Threats to Validity ............................................................................................... 67
   6.3 Future Work ......................................................................................................... 68

Bibliography .................................................................................................................... 69
# List of Figures

2.1 Agent-environment interaction loop in reinforcement learning (adapted from [SB11]). .......................................................... 10
2.2 Non exhaustive list of modern day DRL taxonomies [Res18]. ......... 13
2.3 Schematic Representation: Deep Q-Network ............................ 15
2.4 Schematic Representation: BootDqn Network ......................... 17
2.5 Schematic Representation: Actor critic Network [SB11] ............... 20
3.1 High Level Architecture: Profiling of Agents ............................. 26
3.2 A2C Hybrid Architecture .................................................. 29
3.3 Deep Sea Exploration ....................................................... 32
4.1 Environment loop for interaction between actors and environment [HSA +20 ] 37
5.1 Default behaviour of bsuite agents ........................................ 42
5.2 Radar plot: Learning Rate of default DQN (0.05) vs 0.01 vs 0.1 ....... 43
5.3 Histogram: Learning Rate of default DQN (0.05) vs 0.01 vs 0.1 ....... 43
5.4 Radar plot: Epsilon of default DQN (0.05) vs 0.025 vs 0.1 .......... 45
5.5 Radar plot: Epsilon Greedy DQN vs Boltzmann DQN with Epsilon 0 and 0.05 ................................................................. 46
5.6 Radar plot: Default DQN (Adam Optimizer) vs DQN (Rmsprop Optimizer). 47
5.7 Histogram: Default DQN (Adam Optimizer) vs DQN (Rmsprop Optimizer) 47
5.8 Radar Plot: BootDqn with masks all ones (mt=1) ....................... 48
5.9 Histogram: BootDqn with masks all ones (mt=1) ....................... 48
5.10 Radar Plot: BootDqn agent with 10 and 30 Ensemble heads .......... 50
5.11 Radar Plot: BootDqn agent with 40 Ensemble heads ................. 50
5.12 Radar Plot: BootDqn Noise Scale 0.5 vs 1 ............................. 51
5.13 Radar Plot: A2C with max seq length of 64 ........................... 52
5.14 Radar Plot: R2D2 with LSTM sizes of 10, 20 and 40 ................. 53
5.15 Radar Plot: DQN with drop out rates of 0.2, 0.4 and 0.8 .......... 56
5.16 Histogram: DQN with drop out rates of 0.2, 0.4 and 0.8 .......... 56
5.17 Radar Plot: Default DQN vs Noisy DQN ............................. 57
5.18 Histogram: Default DQN vs Noisy DQN ............................. 58
5.19 Radar Plot: A2C with network sizes of 32 and 128 ................. 59
5.20 Radar Plot: Hybrid A2C ................................................... 61
5.21 Radar Plot: Hybrid A2C (Linear Layer) ................................ 62
5.22 Radar Plot: Hybrid R2D2 .................................................. 63
5.23 Histogram: Hybrid R2D2 .................................................. 63
1 Introduction

We begin the master thesis i.e. "Performance Profiles of Basic, Memory-Based and Hybrid Deep Reinforcement Learning Agents with Bsuite" with the introduction chapter. We have divided the Introduction chapter into 3 sections where the foremost section is Motivation. Here, we describe the motivation behind our research, where we describe the RL domain and the research going into the field, what are the problems researchers are facing due to modern developments, and how do we propose a solution to address these problems. The second section is the Main Contributions, where we discuss in brief the contributions of our work. Finally, we go to the last section of this chapter called the Structure of the thesis. In this section, we give a basic idea of the structure of this thesis, so that the readers are not only aware of the rest of the chapters of our thesis, but also what are the contents of the respective chapters.

1.1 Motivation

In this section, we will be discussing the motivation behind our thesis and research questions. So, from the early times, Reinforcement Learning (RL) had to deal with RL problems, which were usually associated with the RL algorithms trying to resolve the RL problems by interacting with the environments. The ultimate goal of the RL algorithms is to maximize the overall cumulative rewards[FIHI+18] [SB11]. When compared RL to other domains of control it differed in a way as RL did not have the overall idea of the environment. Also, unlike the other branches such as Statistics or Machine Learning the action taken had an effect on the future rewards. An RL algorithm or agent has to display its capabilities on basis of its learning abilities from the interaction. The agent interacts with the environment and takes action, later using these experiences agent learns how to take action and perform better. This is the approach taken by the RL agent to solve the RL problem.

In recent times, the research in the field of development of RL algorithms has been increased to a great extent. According to some definitions, there are a wide variety of tasks that RL can solve with human-level performance or even above that, such a wide range of tasks solved by RL is categorized under general artificial learning (AGI) [Min61][LH07]. Researchers in the field of RL has accomplished a great deal on a variety of tasks which are noted as follows: With the help of RL, researchers can identifying individual digits [LBBH98] and Master the ImageNet classification task [DDS+09][KOQD19]. The RL algorithms have also proven their usefulness and capabilities in playing games and solving tasks such as playing Checkers [Sam59], Backgammon[T+95], Atari Games [MKS+15], defeating the world champions at AlphaStar Go [VBC+19] and more. They have also impressed the community of DRL researchers in the domain of robotics with their abilities to solve robotic domain-related tasks such as Learning from In-Hand manipulation...
as well[OBC+18]. With the ever-increasing accomplishments, the RL algorithms have been getting more and more complex. Some researchers have also started to recognize the potential of the RL algorithms to solve computer-related problems [MNN+19] but there hasn’t been much research and application of RL algorithms in this domain. Few researchers believe [MNN+19] that the reasons for this could be, the real world scenarios being more complex than the traditional environments (with varying action spaces, high dimensional data), and also due to the lack of availability of DRL benchmarks.

With the advent of Deep Reinforcement Learning, there have been numerous agents trying to maximize the learning process and surpass the limitations of the former agents to efficiently solve a variety of problems. With the introduction of each new agent, the agent has been evaluated with respect to a single task or environments like 'Cartpole[BSA83]' and 'Mountain Car[Moo90]' to prove its worthiness. Even the benchmarking tools that were used till now to evaluate these agents were focused on only a single dimension. There had been hardly any research into the field of benchmarking agents over multiple dimensions until recent times. We derive our motivation, in order to address the standardization problem of DRL algorithms and the lack of availability of DRL benchmarks. In this research, we have evaluated the RL agents and created the RL agents profiles based on their performances in carefully designed and specified environments. The environments were designed and selected on the basis of evaluating agents on their core capabilities. We provide agent profiles on performances over various core capabilities like ‘exploration’, ‘memory’ etc. The idea of evaluating the agent’s behavior i.e. agents over multiple dimensions could help us identify the broader areas of problems the agent could be applicable to. It would provide a deeper insight into the RL agent’s performance across the core capabilities and thus help the researchers understand the applications of RL agent w.r.t to the RL problem. Evaluation of agents on different dimensions also open a lot of opportunities in hyperparameter tuning and we can visualize how certain hyper-parameters tuning can affect agents behaviour across certain dimensions. These insights could help us to tailor the tuning of agents to solve various tasks. One more area in which multidimensional benchmark marking could contribute is in the selection of an agent among a set of agents to perform specific tasks by comparing the different agents across different dimensions and choosing that has a higher performance in the dimension required to solve the specific task[OHA+20].

Bsuite[OHA+20], is an RL-based benchmarking tool that comprises various experiments designed to test the performance profiles of agents over multiple dimensions. Bsuite is a standalone state-of-the-art evaluation tool, which evaluates the DRL agents based on their behavior across a set of environments. The knowledge of the agent’s profile would provide us deeper insights into the agent’s core capabilities and allow us to reap all the above-mentioned benefits. With the help of the agent profiles, we can not only understand the agent’s architecture but also the influence of the hyperparameters on agents’ performance across various tasks. Hence, we intend to address the same in terms of scientific research questions mentioned in depth in the Chapter 3, about how much influence the hyper-parameter have on RL algorithms, to what level do the hyperparameter tuning impacts the performance of agents, and can hybrid agents be created using a combination of components from various RL algorithms. We aim to use Bsuite, to create various agent profiles, so that agents are evaluated based on their core capabilities, weigh the impact and influence of hyperparameter tuning and architecture changes by comparing perfor-
mance profiles of agents, and encourage hybrid agent development based on the agent profiles.

Hence, in our thesis topic, we are addressing the problems in the standardized evaluation of the DRL agents and generate performance profiles for a series of Basic, Memory-based agents. Check the influence of various parameters on RL algorithms and encourage the development of hybrid agents highlighting their performance profiles and contributing more in the field of 'Evaluation of DRL' agents and 'Bench Marking of DRL agents'.

1.2 Main Contributions

This is the second section of Chapter 1, in this section, we discuss the major contributions from our work.

- We aim at performing a thorough literature survey of the benchmarking tools available for the evaluation of the DRL agents and RL algorithms.
- We generate agent profiles of a variety of Basic and Advanced DRL agents.
- We use simple agent implementation majorly from two frameworks viz. Bsuite[OHA+20] and Acme[HSA+20].
- We hyper tune a series of parameters from all the lists of basic and advanced DRL agents in our master thesis. We generate a series of agent profiles of these hyper tuned agents and compare them with the default DRL agent profiles in order to learn the influence and the extent of influence each hyperparameter posses on the RL algorithm.
- We make changes in the architectures of various DRL algorithms and again generate their profile. Comparison of these profiles with their default profiles helps us validate our hypothesis and record new findings to publish in the field of DRL agents profiling. This also helps to understand the different components of the architecture and its emphasis on the performance of the DRL algorithms.
- We further attempt to learn the important components of the different DRL algorithms and use a combination of these components to create a powerful agent with a better performance profile. Hence, encouraging the development of hybrid agents based on the knowledge of the DRL algorithm and agent performance profiles.
- Finally, we share our findings and the various basic, advanced, and hybrid DRL agent’s performance profiles for the RL community as a foundation for further future work in this domain.

1.3 Structure

This is the third and final section of this chapter. Here, we will see the whole structure of the master thesis and the remainder of the chapters as well as the contents discussed in the remainder of the chapters.
• **Background:** This chapter gives the readers a basic background related to our master thesis. It starts with a literature overview, describing the strategy and method to extract resources relevant to our master thesis. In the second section, it gives a general background about the basic concepts on Reinforcement Learning, Classification of RL algorithms, later we narrow down it to the DRL algorithms and their definitions which are used in our thesis, and the final section providing information about the related work w.r.t to our benchmarking of DRL agents.

• **Design:** In this chapter, we provide an overall design of our master thesis. This chapter is again divided into two major sections viz. Research Questions and Design. In the first section, we briefly define the research question we are scientifically addressing in this work. In the second section, we go in-depth and explain the different components used in the design of this thesis work.

• **Experimental Set-Up:** This chapter aims at providing details about the software versions of different tools, technologies, and frameworks used in our work. We divided this chapter into two sections, the first gives a deeper idea about the DRL frameworks used and the second section provides the version, configuration settings, and details about the code components and their structure in order to facilitate the reproducibility of results.

• **Evaluation and Results:** This is one of the most important chapters in this report. As the name suggests, it contains the evaluation of our experiments and their respective findings. Here, the list of experiments is grouped in terms of their research questions, and the experiments are described in detail. Each experiment in this contains 3 sub-sections namely. Set-up (Giving the parameter and configuration settings), Hypothesis (The underlying hypothesis behind the experiment), and Discussion (discussion about the findings of the experiment).

• **Conclusion and Future Work:** This is the final chapter of our master thesis. In this chapter we conclude our experiments, evaluation, results, and findings, we mention the different threats that are needed to be validated and the future work and its scope related to our master thesis.
2 Background

In this section, we briefly discuss the background related to Deep Reinforcement Learning. We will dive into the introduction of reinforcement learning, discuss various basic concepts, DRL algorithms, and then switch to literature overview where we discuss a bit about the resources we used to give the whole literature overview about the evaluation, benchmarking, and profiling of agents.

This chapter consists of the two major sections namely viz. Introduction to Reinforcement Learning and Literature Overview. Let’s focus on the contents of the first section i.e. Introduction to Reinforcement learning, this section is further sub-divided into subsections like Introduction to DRL, Introduction to various DRL methods, Classification of DRL methods, and finally discuss the underlying architectures and concepts of different DRL algorithms used in our thesis. In the second section, we discuss a little literature survey we did on the topics such as evaluation, benchmarking and profiling of agents. Here, we discuss the different methods we used to fetch materials related to our work and present the related work in this field. The structure and the contents of this chapter are given as follows:

- In Section 2.1, we start with the introduction to Reinforcement Learning (RL), Deep Reinforcement Learning (DRL), the various families of DRL methods, challenges of DRL, and also discuss the various benchmarking nuances related to DRL agents.
- In Section 2.4, we will briefly discuss the structure and overview of the literature. Here, we mainly discuss the various sources and methods we have used to run a literature survey on the background of deep reinforcement learning (DRL). Due to the scarcity of literature, we have to run a literature overview about the various topics such as evaluation techniques, benchmarking tools, and profiling of agents.

2.1 Introduction to Reinforcement Learning

Machine learning is always known for its automated methods and its capabilities to find different patterns in the data to solve various tasks, these tasks can be divided into 3 types:

- Supervised Learning: In this type, tasks are associated with drawing inference from labeled data.
- Unsupervised Learning: In this type, tasks are associated with drawing inference from unlabelled data.
- Reinforcement Learning: This task deals with learning how agents interact with the environment to maximize cumulative rewards.
The idea that individuals without prior knowledge can learn something about the environment without the help of a teacher by only interacting with the environment gave rise to the application of this concept into machine learning which was known as reinforcement learning[SB11]. Reinforcement learning is a field of machine learning that focuses on goal-oriented learning of individuals from interactions with an environment. It consists of two majorly components namely, agents and environment. The agents are nothing but the individuals that interact with the environment and based on the interaction they receive a numerical signal called a reward. The goal of reinforcement algorithms is to maximize these rewards whilst interacting with the environment[FlHI+18][SB11]. To elaborate on this, the environment consists of a given set of states and the agent interacts with the environment by taking some action as a result of which the environment would transition to the next state providing some reward based on the action. The agents repeat this series of interactions to maximize the total reward and hence learns to solve the Reinforcement Learning tasks incrementally. In this section, we will particularly focus on mostly the RL agents that use neural networks in their definition and hence the name Deep Reinforcement Learning agents (DRL agents). Furthermore, we would only focus on the RL and DRL concepts and algorithms that are relevant to our work.

![Agent-environment interaction loop in reinforcement learning (adapted from [SB11]).](image)

- **Markov Decision Process:** As discussed above the idea of agents and environment was derived from the concept called as Markov Decision Process. This process is comprised of the following components viz. agents, environment, states, actions, and rewards. Problems are defined using finite Markov Decision Processes (MDP) [SB11] [FlHI+18] [Resa]. The process is defined as a 5-tuple \((S, A, T, R, \gamma)\) where \(S\) is the state space, \(A\) is the action space, \(T: S \times A \times S \rightarrow [0, 1]\) is the transition function describing transitional probability between states, \(R: S \times A \times S \rightarrow [0, R_{\text{max}}]\) is the reward function and \(\gamma \in [0, 1)\) is the discount factor. The MDP property states that the future state depends on the current state and not the history of past states. There is a finite set of states \(S\), actions \(A\), and rewards \(R\). At each time step \(t\) the environment is in state \(s_t\) and the agent observes this state and choose an action from a possible set of actions available this in return shifts the environment to state \(s_{t+1}\) giving reward \(r_{t+1}\) to the agent. The \(S_t\) and \(R_t\) are random variables that have well defined probability distributions. That is, all the possible values that are assigned to this variable have some associated probability. These distributions depend on the preceding state and action that occurred in the previous time step[Resa]. The rewards selected by the agent are called the expected returns and
the goal of the agent is not only to maximize the immediate reward but also to maximize the cumulative reward to get the maximum expected returns.

The learner in the MDP problem is called the agent, the problem it is interacting with is called the environment. Hence, the agent does not have the complete idea of the environment it interacts with the environment and selects actions as per its policy on a trial and error basis to complete the task with maximum returns.

• **Expected Return:** To achieve the goal of maximizing the cumulative reward, the concept of expected return is used to aggregate or formalize the rewards at a particular time step. When there is a time step that starts from \( t_0 \) till \( T \) we have a final time step \( T \). The agent environment interaction is broken down into subsequences called episodes (Eg: Each round of a game where the terminal state, final time step \( T \) is reached when a player scores a point) [Resa]. After each round of a game, the environment is set into a reset \( s_t \) some starting state or a random sample from a distribution of possible starting states and then new episodes start. A task related to an episode is called an episodic task. The tasks where there are no limits and no episodes and the tasks continue without breaking are called continuous tasks. In this report, we are only focusing on discrete tasks.

\[
G(t) = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.1}
\]

Now we change our goal to maximize the expected reward to discounted expected reward. It requires a discount rate \( \gamma \) defined between 0 and 1 and it discounts the future rewards and finds the present value of future rewards. (By this our agent would care more about the immediate reward than the future reward as future rewards would be heavily discounted).

• **Policy and Value Functions:** A Policy defines how likely it is that an agent will select a specific action given a specific state [Resa] [Res18] [SB11]. The policy is denoted by \( \pi \) which is used by the agent to determine what action is to be taken given the environment’s current state. Value Functions decides how good is a specific action or a specific state for the agent [Resa] [Res18] [SB11]. How good a state is or how good an action is given the state is given in terms of expected return. Rewards help us to find the quality of the intermediate decisions, Similarly, value functions help us to identify the total rewards we can get when choosing some actions. The state-value function for policy \( \pi \), denoted as \( V^\pi(s) \), tells us how good any given state \( s \) is for an agent following policy \( \pi \). In other words, it gives us the value of a state \( s \) under \( \pi \). The value of state \( s \) under policy \( \pi \) is the expected return starting from the state \( s \) under policy \( \pi \) at time \( t \).

\[
V^\pi(s) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, \pi] \tag{2.2}
\]

The action-value function for policy \( \pi \), denoted as \( Q^\pi(s, a) \), tells us how good it is for the agent to take any given action \( a \) from a given state \( s \) while following policy \( \pi \). In other words, it gives us the value of action \( a \) under policy \( \pi \). The value of action \( a \) in the state \( s \) under policy \( \pi \), is the expected return starting from the state \( s \) with action \( a \) under policy \( \pi \) at time \( t \). The output i.e. the expected value is also
called as Q value. The action-value function also sometimes known as the quality function is defined as:

\[ Q^\pi(s, a) = \mathbb{E}\left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a, \pi \right] \]  

(2.3)

- **Optimal Policies**: The goal of the reinforcement learning algorithm is to seek a policy that would the max expected return to the agent than the other policies and this policy is nothing but optimal policy. An optimal policy has a greater or equal expected return to other policies for all states. The optimal policy has an optimal state value function. The optimal value function \((V^*)\) gives the maximum expected return achievable by any policy for each state.

\[ V^*(s) = \max_\pi V^\pi(s) \]  

(2.4)

Similarly, there is optimal q function \((Q^*)\) which gives the maximum Q value achievable by any policy for each possible state, action pair, and it is defined as:

\[ Q^*(s) = \max_\pi Q^\pi(s, a) \]  

(2.5)

One of the advantages quality functions have over value functions is that the optimal quality functions can informally used to find optimal policy as:

\[ \pi^*(s) = \arg\max_a Q^*(s, a) \]  

(2.6)

Many of the reinforcement learning algorithms focused on finding optimal quality function as it would eventually derive the optimal policy. Hence, the family of algorithms started focusing on finding optimal quality functions to get the optimal policy came to be known as Value-Based algorithms.

To summarize the above, we can state that reinforcement learning is formalized in the MDP process consisting of agents, environment, rewards, etc. There are few agents that focus on learning the model and others that are model-free agents out of which we focus on model-free agents. The agent interacts with the environment to maximize the cumulative rewards. Various functions like state value function, action state value functions are then used to find the optimal policy. The optimal policy found in turn tells us which action to take in the corresponding observed state to get maximum cumulative rewards. Also, the optimal quality function could be used to directly derive the optimal policy. Finally, the algorithms that follow this property are known as Value-Based algorithms.

### 2.2 Classification of Reinforcement Learning Algorithms

Till now, we saw the basic concepts and terminologies in Reinforcement Learning and Deep Reinforcement Learning. Now let’s move onto the next topics, where we discuss the different types of DRL algorithms and their taxonomies. Also, I want to mention a disclaimer before we begin this section. As mentioned in the [Res18] also itself that it
is difficult to draw a quite accurate, all-encompassing taxonomy of the DRL algorithms. It is also difficult to get an accurate representation of the taxonomies due to the modular nature of the algorithms and not be able to be represented in the tree structure. Nonetheless, I have used the taxonomy provided by the [Res18] for the reference in this section.

![Figure 2.2: Non exhaustive list of modern day DRL taxonomies][Res18].

As we can see in the Figure 2.2 the modern-day RL algorithms are majorly branched on the basis of the environment knowledge. To further clarify this point, the DRL agents are classified based on their knowledge or access to the model of the environment. The model of the environment comprises the state transition function and the reward function [Res18]. In cases where the DRL algorithms or the agent has knowledge about the state transition function and the reward function, it has prior knowledge of the environment. When the agent already has the knowledge of the environment it can plan ahead before selecting an action as it knows the state transition function and the reward function, it can compute all scenarios of taking different actions and computing the expected return associated with each action. The disadvantage with this approach is that most of the time the knowledge of the environment is usually not available. In many scenarios, the ground truth of the environment is not available, in such cases, the agent has to rely on experiences to learn the model. Again, the problem with this approach is the model bias which would be exploited by the agent. In such circumstances, the agent would perform well on the model it is trained on but it would perform poorly on other real-world-based environments. Hence, because of all these reasons Model learning is very hard and most of the researchers perform research with the model-free agents[Res18]. To summarize all of it, the agents who use models are model-based agents, and the agents who don’t are model-free agents.

- **Model Based vs Model Free:**

As discussed above the wide variety of reinforcement learning algorithms are divided into two sections i.e. Model-Based and Model-Free

- **Model-Based:** The agents here, tend to learn the environment in order to predict the behavior of the environment. By learning the environment agent can predict the future states and rewards given the current states and actions.
This type of agent is out of scope for our research hence we pay our attention to the other section.

- **Model-Free**: These agents do not learn the environment as it could lead to many problems like introducing model bias by learning the model, knowing the ground truth about the environment, etc. Hence these agents learn from their experiences that are generated during interactions with the environment.

In our work, we have only used model-free agents. Hence, in the below sections we would be mostly focusing on the concepts and algorithms related to the model-free DRL algorithms.

- **On Policy vs Off Policy**:

  After the DRL algorithms being classified as Model-based and Model-free, there is further classification. Again, we would only focus on model-free branching. As shown in the Figure 2.2 we can see that the model-free agents are further branched as either On Policy or Off Policy. Therefore, below we have mentioned the two concepts namely: On Policy and Off Policy.

  - **On Policy Algorithms** The algorithms that lie in this category actually work on improving policy performance. They do not store past data and then perform learning rather they try to achieve better performance on policy.[Wie04] All the algorithms under this section follow policy and hence the name On-Policy as they always follow policy and only take on policy data to perform the updates. Vanilla Policy gradient, TRPO, and PPO are algorithms that belong to this family.

  - **Off Policy Algorithms** The algorithms that are classified as off policy operates differently than on policy ones. These algorithms do not always follow policy hence the name Off Policy. They usually work on the concept of Q Learning and most of them aim to learn Q functions to update the policy. The algorithms in this category make efficient use of old data to learn Q functions, this Q function is then used to improve the policy

### 2.3 Deep Reinforcement Learning Algorithms

Till now, we have seen the basic concepts of RL, DRL, and the different terminologies related to the classification of the DRL agents. Here, we have a dive into the different DRL algorithms. As already mentioned in the previous section that in this work we mostly deal with model-free agents. Furthermore, we will only look into those algorithms which have been used in our work.

#### 2.3.1 DQN

In this section, we mentioned one of the first RL algorithms introduced by [Hua20]. DQN is one of the first RL algorithms which uses a Deep Neural Network in its underlying architecture to calculate Q-values for each of the actions available in the environment[SB11]. The researchers at [Hua20] have also tested the performance of this DQN agent on a
series of Atari 2600 games where it was able to attain a performance even better than the human level. This has encouraged the use of Neural Networks in the development of the RL agents. In [Hua20] the authors have used video frames of the games as the input to the Neural Network which then produces an output of Q-values per action. The underlying schematic of the network is depicted in the Figure 2.3 where we can see that the environment state is passed as inputs to the deep neural network (Which is comprised of a Convolutional neural network (CNN)) and the neural network then approximates Q values for each of the possible actions. In this setting, the deep neural networks learn using the stochastic gradient descent approach to minimize the loss and update its weight parameters based on this loss. The underlying equation DQN uses as its loss function is given in the Equation 2.7 is as follows:

\[
L = \left( r + \gamma \max_{a'} Q(s', a') - \underbrace{Q(s, a)}_{\text{prediction}} \right)^2
\]  

Figure 2.3: Schematic Representation: Deep Q-Network

In the literature, the researchers [TVR96] have highlighted the issues reinforcement learning faces when combined with a neural network. As per the authors of [TVR96] when RL algorithms combined with neural networks not only makes the DRL algorithm unstable but also faces problems converging into a global optimum. Due to such issues of the DRL algorithm does not converge, there was not much development in the field of DRL algorithms until DQN solved all these problems. With the super human-level performance at Atari games [Hua20], DQN showcased the potential of the use of deep neural networks in reinforcement learning. Let’s now see how DQN solves the above problem. DQN uses a couple of major strategies like the use of a ’Replay Buffer’ and ’Target Network’ to solve the above issues.

- **Replay Buffer**: During the learning process, the agent interacts with the environment based on its knowledge of the state of the environment and it receives a reward. All this information is stored in a replay buffer which later can be used
to train the agent. Hence, the agent learns from the experiences stored in the replay buffer. Now, during learning the most recent experiences from the replay buffer are not fetched, rather DQN uses a mini-batch of experiences in a random manner from the buffer. This random mini-batch approach cuts down the strong correlation between the experiences when fetched on the basis of the most recent experiences\[SB11\] [FIHI+18].

- **Target network:** DQN uses another component called a Target Network. The need for this component arises from the traditional use of a single network. As earlier a single network was used for learning from its experiences as well as making predictions based on the observations received, this created a problem of moving targets. Because the single network is making predictions and learning at the same time, the learning network is also moving with each update, and this created issues in converging of the network as the target network are also moving. Hence, DQN introduced another network along with the traditional network called as target network[SB11]. In this setting, the single network only makes predictions based on the observations from the environment and the target network learns from the experiences. Here, the target networks weights are updated every $C$ time step, and the base network weight is updated at each time step. After the agent has run for $C$, the weights of the base network are updated with the target network’s weight and new targets are set. With the help of this strategy, DQN overcomes the problem of moving targets and helps the DRL algorithms to converge better.

The DQN algorithm mentioned below incorporates all the above-discussed ideas of replay buffer and target network:

**Algorithm 1:** DQN network with replay buffer and target network [Hua20]

1. Initialize replay memory $D$ to capacity $N$
2. Initialize action-value function $Q$ with random weights $\theta$
3. Initialize target action-value function $\bar{Q}$ with random weights $\bar{\theta} = \theta$
4. for episode $\leftarrow 1$ to $M$ do
5. Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6. for $t \leftarrow 1$ to $T$ do
7. With probability $\epsilon$ select a random action $a_t$
8. otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
9. Execute action $a_t$ in emulator and observe reward $r_t$ and next observation $x_t$
10. Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
11. Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
12. Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
13. Set $y_j = \begin{cases} r_j, & \text{for terminal at step } \phi_{j+1} \\ r_j + \gamma \text{max}_{a'} Q(\phi_{j+1}, a'; \bar{\theta}), & \text{for non terminal at step } \phi_{j+1} \end{cases}$
14. Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
15. Every $C$ steps reset $\bar{Q} = Q$
16. end
17. end
2.3.2 BootDqn

In this section, we have mentioned in-depth the underlying architecture of the BootDqn agent [OBPV16]. In the field of RL, exploration is always one of the core capabilities required by the RL algorithm in order to solve complex RL problems. In the RL setting, the agent does not have an idea of the environment and it learns from its experience. So for the agent to effectively learn and maximize the cumulative rewards. Each action taken at an earlier stage can have an effect on the cumulative rewards so the agent has to maintain a balance between exploration and exploitation [SB11] [FIHI’18]. The agent needs to effectively learn by exploring the states and collect informative value about the states of the environment and hence exploration is always a desired quality in RL algorithms.

In the literature of RL algorithms efficient exploration has always remained a problem. Traditionally in RL, the $\epsilon$ greedy method was used as a dithering strategy for exploration which led to exponential data requirements. There was not much attention put on the temporally extended or deep exploration [OBPV16]. Most of the algorithms were dependent on the statistically efficient RL approaches which were designed in low dimensional environments with low action spaces so these approaches cannot be applied for complex RL problems. So, there was always a need for an exploration strategy in the RL algorithm. Hence, alternative approaches for effective exploration were used to solve this problem. The researchers at [OBPV16] used an effective approach called a randomized prior function which was inspired by Thompson sampling. Unlike in the tradition $\epsilon$ greedy approach where the action was picked on the basis of the highest estimate value, these alternate approaches maintained a distribution over possible values and enabled exploration by randomly selecting a policy to select an action. Hence, they introduced bootstrap DQN with such alternate approaches for introducing reasonable amounts of uncertainty in the neural network for leveraging deep exploration.

![Figure 2.4: Schematic Representation: BootDqn Network](image)

The introduction of the BootDqn algorithm enabled the deep exploration and improved its performance over most games [OBPV16]. The authors of [OBPV16] was used BootDqn
along with random initialization and achieved computationally good performances in the field of exploration-related RL problems. The main principle of this algorithm is to approximate a population distribution from the sample distribution. It takes a data set $D$ and an estimator $\psi$ and creates a bootstrapped sample $\bar{D}$ with the same cardinality as $D$ by uniformly sampling from $D$ with replacement. Finally, the estimate of the bootstrap is given by $\psi(\bar{D})$. This bootstrap estimate is then used as a representation by the heads used in the BootDqn architecture. So, let’s now focus on the BootDqn architecture. The underlying schema of the BootDqn is shown in the Figure 2.4, where we can see that it uses input as the environment states to the shared network and the shared network has multiple heads on its output. These heads are called bootstrap heads and they are branched independently of each other. Each bootstrapped head is trained on its bootstrapped sub-sample of the data and is represented by $\psi(\bar{D})$. The shared network learns a joint feature representation across the entire data set. Such types of networks can be trained easily with forward/backward pass as each head is independent of the other. So, let’s discuss the major components of BootDqn architecture to get a better understanding of its behavior. The major components of the BootDqn which are discussed in the below sub-sections are ‘K-Heads’ and ‘Mask’.

- **K Heads:** As discussed above and also shown in the Figure 2.4, there are K heads in the architecture of the BootDqn agent. These heads are called Bootstrap Heads. Each Head takes its input from the underlying shared networks. Now, there could be two sets with the underlying shared network, either it could be a shared network with multiple k networks calculating the $Q_k$ value functions or there could be a shared network with k heads representing $kQ$ value functions. The BootDqn algorithm described below satisfies both cases. So the idea behind K heads is that each head can take a bootstrapped data sample from the shared network and estimate its q value function, where each K head is data-dependent and learns a specific value function. Hence, k heads learn k different Q value functions and enable effective exploration. They also use a mask on top of the heads to filter out a subset of K heads during each iteration and which helps in achieving deep exploration.

- **Mask:** As discussed above the BootDqn agent uses a mask on top of its k heads to restrict the no. of k heads during training. this mask decides which k heads should be enabled and learn during the training phase at the t time step. So, the mask is nothing but a binary vector of length of K containing a value corresponding to each k head of the agent. There are two modes in the mask viz. Binomial mask and Poisson masks. In the binomial mask setting the mask of length randomly, get a value between 0 or 1 depending on the Bernoulli’s distribution and called as Double or Nothing Bootstrap. In the later setting, the mask contains all one’s and operates in the ensemble method where every k head is enabled during training. Therefore, masks are implemented on top of the K heads to decide which k heads should participate in the Q value function approximation and this in return gives us a diverse set of K heads having k different Q value function approximators enabling deep exploration

The BootDqn algorithm mentioned below incorporates all the above-discussed ideas of K heads and masks:
Algorithm 2: BootDqn algorithm [OBPV16]

1. **Input:** Value function networks \( Q \) with \( K \) outputs \( Q_k \) where \( k \) ranges from 1 to \( K \), Masking distribution \( M \).

2. Let \( B \) be replay buffer storing experiences for training.

3. **for each episode do**
   4. Obtain initial state from environment \( s_0 \).
   5. Pick a value function to act using \( k \sim Uniform(1, ..., K) \).
   6. **for step \( t = 1 \) until the end of episod do**
      7. Pick an actor according to \( a_t \in \arg\max_a Q_k(s_t, a) \).
      8. Receive state \( s_{t+1} \) and reward \( r_t \) from environment, having taking action \( a_t \).
      9. Sample bootstrap mask \( m_t \sim M \).
     10. Add \( (s_t, a_t, r_{t+1}, s_{t+1}, m_t) \) to replay buffer \( B \).
   11. **end**
4. **end**

---

2.3.3 A2C

In this, we are going to look at the Advantage actor-critic agent. Till now, we have learned about the value-based and policy-based methods in the basic concepts of this section. In value-based methods we learn a value function that will map each state-action pair to a value and at each step, we take an action with the highest value. The second method is the policy-based method (REINFORCE with Policy Gradients [Wil92]), here we directly aim to optimize the policy of the algorithm without using a value function [Wil92] [Resb]. The problem with Policy Gradients is that we are in a Monte Carlo situation, where we have to wait till the end of the episode to calculate the reward. The main problem with such an approach is we only check the reward at the end of the episode. So, if there is a high reward we state that all the actions in the episode are good even if a few of the actions taken are bad, we only see the end cumulative reward. As a result of this problem, in order to have an optimal policy, we require to take a lot of samples. Actor critic algorithm [Wil92] has a better scoring as compared to the above approach. Unlike the Monte Carlo, we can update the network at each time step also known as Temporal Difference Learning.

In the section, we focus on the actor-critic agent which is a Temporal Difference version of the Policy Gradients. Since we are updating the model at each time step we cannot use the total rewards \( R(t) \) in the update section rather we use the Q function. Now, let’s have a look at how the actor and critic function work. As shown in the Figure 2.5 there are two separate memory structures to independently represent the policy and the value function. The network representing the policy is called as ‘actor’ and the one representing the value function is called as ‘critic’.

- **Actor:** The actor network selects the action at each time step based on it’s policy. Based on the critic feedback the actor updates it’s policy parameter \( \theta \) for \( \pi_\theta(a|s) \).

- **Critic:** The critic network critiques, how good is the taken action. Critic updates the value function parameters \( w \) and depending on the algorithm it could be action-value \( Q_w(a|s) \) or state-value \( V_w(s) \).
The networks in the actor-critic model use an Advantage function to update the actor losses in order to train the A2C agent. The advantage is defined as follows:

$$ A(s, a) = (r + \gamma V(s') - V(s)) $$

(2.8)

The advantage function helps in reducing the variance in the RL algorithms. The advantage function tells us the improvement as compared to the average value of the state. When $A(s, a) > 0$ the gradient is pushed in that direction, and when the $A(s, a) < 0$ that means the action selected does even worse than the average and the gradient is shifted in the opposite direction. In the Equation 2.8 we use TD error as a good estimate for advantage function.

Let us now look at the pseudo code of the simple actor critic algorithm [Lil18]:

### 2.3.4 R2D2

Here, we discuss in brief the Recurrent Replay Distributed DQN (R2D2) agent. We also discuss the different components and strategies used by the R2D2 agent [KOQD19], in
Algorithm 3: Actor critic algorithm [Lil18]

1. Initialize $s, \theta, w$ at random; sample $a \sim \pi_\theta(a|s)$.

2. for $t = 1..T$ do

   3. Sample reward $r_t \sim R(s_{t,t})$ and next state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$;

   4. Then sample the next action $a_{t+1} \sim \pi_\theta(a_{t+1}|s_{t+1})$;

   5. Update the policy parameters: $	heta \leftarrow \theta + \alpha_\theta Q_w(s_t, a_t)\nabla \theta \ln \pi_\theta(a_t/s_t)$;

   6. Compute the correction (TD error) for action-value at time $t: \delta_t = r_t + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)$

   7. And use it to update the parameters of action-value function:

   $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s_t, a_t)$

end

order to understand its behavior. Due to the recent advances in the distributed training of RL algorithms and the need for memory-based agents, there is more research going on in the field of RNN based RL agents. R2D2 is one such agent which makes use of the modern-day components such as recurrent states, experience replay and distributed training [KOQD19]. R2D2 is the first agent which has achieved a feat of human-level performance in 52 out of 57 Atari games. Due to its performance proficiency over a set of complex RL problems. We have considered R2D2 as one of the main agents in our thesis. The R2D2 agent aims to solve issues related to parameter lag, representational drift, and recurrent state staleness due to the increasing usage of RNN related components in the network of the RL algorithms. It also integrates the findings from the research related to the usage of RNN’s with experience replay and attains state-of-the-art performances in the Atari-57 [BNVB13] and DMLAB-30 [BLT+16].

Due to the advantages of Distributed Reinforcement Learning, modern-day algorithms are increasingly developed based on this principle. One such modern-day distributed training-based RL algorithm is Ape-X [HQB+18]. Ape-X makes use of the decoupling feature, where multiple actors can act and update the experiences in the buffer and the learner can sample random experiences to learn from them. Ape-X also uses n-step targets [SB11], the double Q learning algorithm, The Dueling DQN network architecture, and achieves the state of the art performances on Atari-57 [BNVB13]. R2D2 is most similar to Ape-X built upon n-step targets, double Q learning operating in a setting where typically 256 actors act and generate experiences which are then pulled by a single learner to learn from them. Unlike traditional RL algorithms, R2D2 does not store a transition with a $(s_t, a_t, r_t, s_{t+1})$ rather it stores a sequence of a fixed length of $(s_t, a_t, r_t)$ in the replay buffer with adjacent sequences overlapping with each other by 40-time steps. They differ from Ape-X in terms of their prioritization replay buffer, discount rates, and heuristics. They also change their training process by the addition of a couple of components viz. Stored state and Burn in period [KOQD19]. They undertook this new strategy in order to use the meaningfulness of the long-term dependencies when LSTM is clubbed with experience replay for the training of RNN based RL algorithms. To resolve the issues in the training of RNN based RL algorithms making use of the experience replay they have used stored state and burn-in period, so let’s take a look into these components.

- **Stored state:** R2D2 aims at storing the recurrent state in the replay and use this recurrent state during training. Traditionally, a zero start state initialization has
been used for training and it’s appealing for its simplicity but training RNN based agents on zero initialization does not make use of the meaningfulness of the recurrent state. So, R2D2 aims to use this knowledge of the recurrent state and solve this issue but this does not solve the representational drift and recurrent state staleness.

- **Burn-in:** In this strategy, they have proposed to allow the network to enable a burn-in period where the network takes a portion of the experiences from the replay buffer and burn in those experiences only for unrolling the network and produce a start state. The network can then start updating itself on the remainder of the experiences left in the experiences replay. They also claim that introduction of this feature mitigates the problem of staleness.

Hence, we saw the advances in the field of RNN RL algorithms, the R2D2 performances over the state of the art RL problems and we discussed the various components strategies of R2D2.

This brings us to the end of section Section 2.1 where we saw the basic concepts of DRL, classification of DRL algorithms, and various DRL algorithms used in our master thesis.

### 2.4 Literature Overview

Till now, we saw all the basics related to reinforcement learning, classification of RL algorithms, and various DRL algorithms used in this thesis. Now, we are going to see a short literature overview about the different state-of-the-art benchmarking tools used in the RL community for the profiling and evaluation of RL algorithms.

#### 2.4.1 Methodology

In this section, we mention all the resources and techniques we used as a background for all the sub-sections mentioned under the Introduction to Reinforcement Learning section. So, to begin with, the 'Introduction to RL' we have majorly referred to the [FlHI18] and [SB11] they have provided all the basic information related to RL and DRL. Now, to learn about the classification of DRL agents and the different DRL methods available we used an amalgam of resources including [FlHI18], [SB11], Spinning Up AI website, and the individual scientific papers about the different DRL agents introducing their algorithms. I have also referred to a couple of online blogs to understand the underlying behavior of the various DRL agents used in our work [Resa] [Lil18] [Resb]. Finally, to research the various challenges, benchmarks related to DRL agents we came up with a strategy. We had to derive a strategy to learn about the different benchmarking tools related to DRL agents due to the scarcity of literature and work in this area. As what we are trying to do in our work is already very new and has been less researched on as compared to the other aspects of DRL we did not find many relevant scientific papers related to it. Hence, we had to derive this strategy where we looked up all the relevant papers from the initial 20 search result pages from google scholar. To look up the scientific papers from google scholar we used a bunch of keywords like 'DRL'+'Benchmarks'+'Evaluation'+'Profiling' etc and selected the most relevant papers w.r.t to our work. We also set the advanced
filters to a year from 2015 in order to fetch all the recent benchmarks evaluating the recent DRL agents. In this manner, we decided on different resources and techniques to gather an overview of the literature regarding the background of DRL and its sub-topics.

2.4.2 Literature Survey (DRL Bench marks)

In this section, we discuss the different papers we found related to benchmarking RL algorithms and creating a profile of RL algorithms. So we used the above-mentioned strategy and fetched all the relevant papers either for benchmarking or for the profiling of the DRL agents. We also came across a couple of papers that discuss the problems in the evaluation of DRL algorithms too.

As mentioned numerous times in the section about the advances of the RL algorithm and its potential to provide solutions in a wide range of simple and complex problems. RL algorithms found it’s an application in different fields with high dimensional data, continuous observation spaces, etc. When we searched for the benchmarking tools related to DRL, a few of the benchmarking tools and frameworks we landed upon was in the robotics domain. We mentioned them in our work because we used all the materials available related to the keywords ‘bench marking’ and ‘DRL agents’ to give an overall picture of the availability of the DRL benchmarking tools across different domains. So, researchers have developed benchmarks like Softgym [LWOH20] for manipulating deformed objects, Offworld Gym [KBL+19] for benchmarking RL algorithms on real-world applications like navigation from end to end using robots, Causal World for robotic manipulation of Causal structures, and transfer learning [ATG+20]. Even we found benchmarks that are a bit more relevant but out of our thesis scope because it’s usage of high dimensional spaces or continuous spaces, which are DeepMind Control Suite [TDM+18] where the benchmarking is carried out for continuous control tasks. Safe Exploration [RA19] is also one such tool that aims at the safety of DRL agents when developed for high dimensional spaces. DinerDash [CMH20] is another benchmark created for multi-dimensional spaces with high action dimensions and hierarchical tasks.

When narrowing down to our scope of work the benchmarks were created for particular tasks. Such task-oriented benchmarks evaluate the agent based on their ability to solve a particular task rather than creating a general profile of the agent. Procgen [CHHS19] is one such benchmark consisting of 16 suites of environments procedurally generated to test the abilities of agents in exploration and generalization. Dialogue Management [CBS+17] is another task-specific benchmarking environment that aims at evaluating DRL agents based on Dialogue Management. As mentioned in the above sections that RL can be used in computer-related tasks but not used due to a lack of benchmarking tools. Park [MNN+19] is one of those benchmarks which highlights the problems in RL algorithms and it provides a suite with real-world-like complex environments.

We went through all these papers related to benchmarking tools related to DRL from various domains. None of them evaluated the DRL agents based on creating a general profile of the agent rather they were focused on specific tasks or complexity of the problems rather than giving a general profile. Bsuite [OHA+20] is one of its kind framework which tries to create a general profile of the agent which gives us an idea about the behavior of the RL algorithm. Hence, we had to do a little literature survey to highlight the usefulness
of agent profiling and justifying our selection of a bsuite framework to carry out research related to the master thesis.

This brings us to the end of the Background chapter, which is one of the lengthiest chapters. In this chapter, we saw all the basics, classifications related to RL algorithms, different DRL algorithms, and finally a literature overview about the different benchmarking tools available within the community of RL.
3 Design

In the previous chapters, we saw the basics of DRL, the different DRL agents, and their working principle. Now, in this section, we will focus on the goals that were discussed in the introduction section and define precise research questions that we are going address in our evaluation, a brief introduction about any major software implementation required for the different experiments, and finally describe the environments proposed in the Bsuite Paper.

3.1 Research Questions

This research is the first attempt at profiling deep reinforcement learning agents. We acknowledge the effort behind the bench-marking of DRL agents on certain tasks like 'Cart-pole' Or 'Mountain-Car' but there have been only a few attempts to evaluate the agents on multiple dimensions like Memory, Generalization, Scale, Basic, Exploration, and Noise. In the Bsuite paper, the authors have defined a framework to benchmark different DRL agents on these multiple dimensions but they are restrictive to certain default agents with simple definitions. We use the same Bsuite framework to not only extend its capabilities to different definitions of the agents outside Bsuite’s provided default value-based and policy-based agents but also learn the impact of different hyper-parameters, architectural components of the DRL on the multiple dimensions of benchmarkers defined in the Bsuite. This work of profiling different DRL agents, in particular, addresses the following Research questions:

1. What is the influence of hyper-parameters for basic value-based agents, and to what extent do the hyper-parameters affect the profile of agents?
2. What is the influence of architecture for basic and advanced value-based agents, and to what extent do the changes in architecture affect the profile of agents?
3. What are the best aspects of an agent and If we build a hybrid agent using a combination of best aspects from different agents would it enhance its performance over different benchmarking dimensions?

3.2 Design: Profiling of Agents

In the above section, we mentioned the 3 major research questions which we scientifically address in our work. In this section, we will discuss the overall structure of the profiling process. The structure of the profiling process is shown in the Figure 3.1. As shown in the Figure 3.1, the Design comprises of agents namely (default agent implementation
from Bsuite and Acme), a list of experiments, the Bsuite Environments, and the agent profiles.

In this work, we use different agents from the Bsuite and Acme frameworks. Bsuite framework has already provided us the default profiles of the agents. We use these profiles as the baselines of the agent and make changes in the default definitions of the agents to again extract the agent profiles. The changes carried out on the agent are based on the tunable parameters (based on the hypothesis, which we are trying to validate) and the tunable parameters are considered circumventing the above-mentioned research questions. Finally, comparing the baseline profiles and the newly generated profiles of the DRL agents we can validate the hypothesis of the experiments. By comparing the two profiles, we can also observe the difference in the dimensions and get an intuition about the influence of various hyper-parameters on different agents. Hence, this is the whole profiling process that we have undertaken in this work, encircling our research questions and sharing our findings. In this section, we will briefly go through the components of the Design (Profiling process of agents). As the default definitions of the different agents used in this experiment are already mentioned in brief in the Chapter 2 Section 2.3, to avoid repetitions we haven’t mentioned the agents in this section. We would majorly focus on the list of experiments and the details of the Bsuite environments in this section.

### 3.2.1 Tunable Parameters

In this section, we will have a look at different tunable parameters and the theory behind those parameter tunable experiments. We will go through each agent and the parameters tuned related to those agents. In this section, we will define the changes we do to the hyperparameters and the architecture. As we already covering the hypothesis behind each and every hyper-parameter in the Chapter 5 we will focus more on the architecture aspect of the experiments. We have divided this section based on the general and specific hyperparameters tuned as per the agent. So, let us take a deep dive into the agents and their respective hyperparameters.
• **DQN:**

We use the DQN agent as a baseline agent for the value-based family and we change various hyperparameters common to all value-based agents.

1. **General Hyperparameters:** This experiment is based on testing how the various innate hyperparameters impact the agent and to what extend. The hyper-parameters were tweaked in an attempt to improve the performance of default DQN in the exploration dimension and also to observe the impact of the hyper-parameter changes in the other dimensions. The hyperparameters that were tweaked in this work are mentioned as follows: Learning rate (learning rate of the agent), Epsilon (related to the exploration of the agent) and neural network dropout (to increase exploration) Optimizer (related to the exploration of the agent). Again we do not explain the role of each hyperparameter, as each of the general hyperparameters is further explained in brief in the hypothesis section of Chapter 5.

2. **Boltzman:** In this experiment, we attempt to boost the exploration of the DQN agent by letting it select the actions at random rather than the maximum q-values which is its default behaviour. The default DQN setting operates at Epsilon greedy strategy. In this strategy, the actions are selected at random i.e. exploration till a specific cut off value is reached by the epsilon parameter. The main purpose of the epsilon parameter is to maintain the balance between exploration and exploitation. So, in its default setting the DQN agents explores till a certain value of epsilon and then it starts exploiting the best options. Hence, we attempt to change this behaviour by using the Boltzman approach, where we normalize the actions and rather than selecting the best action as per the old approach i.e. exploitation, now we select the action in random as per our Boltzman approach to maximize the exploration.

3. **Noisy Agent:** This experiment attempts to improve the noisiness in the agent and boost the exploration. In this experiment, we added noisy layers on top of the general neural network. The idea was taken from noisy networks for exploration where we used factorised Gaussian noise. In Factorised Gaussian noise each input unit and output unit are infused with independent Gaussian noise. We use this concept in our DQN agent definition by adding a noisy layer on top of the neural network of the agent so that it introduces more exploration[FAP^18]. So, we add a noisy layer on top of the default DQN to train the agent for different environments, we hypothesize that the addition of the noisy layer would make the network more robust to noise and it would improve the generalization learnt by the agent.

• **BootDQN:**

In this set of parameter tunable experiments, we use the BootDQN agent which already has better exploration than the DQN because of its behaviour to find the distribution of Q values using its bootstrap concept as compared to DQN which only attempts to find the Q values and tries to select the optimum from those[OBPV^16]. In the experiment, we attempt to further increase the exploration for BootDQN agent by performing the below experiments:
1. **K heads**: The Bootstrap DQN agent had multiple neural networks or multiple heads in its definition to effectively implement Deep Exploration. These heads were used to explore different directions in the solution space in order to find an optimum solution which in return also improves explorations. The agent uses the strategy to select one head at random for each episode and try to exploit the head it has chosen during the course of the episode. By default, the maximum exploration was achieved by the total number of 20 heads[OBPV16]. We attempted to tune this hyperparameter to get a higher exploration than the default behaviour.

2. **Ensemble voting**: The Deep exploration via bootstrapped DQN paper also mentions the binary mask that is attached with each tuple of the experience stored in the experience replay buffer. The experiences stored in the buffer is being used by the multiple neural networks in the bootstrapped DQN agent to train themselves. In their default setting the agent uses the independent Bernoulli’s distribution to initialize the mask to achieve randomness during training[OBPV16]. Whereas, there was an alternate use of the mask which when set to all 1’s changes the training to ensemble voting method. We changed the training method to the ensemble to observe the change in the behaviour of the agent and to see the impact of this hyperparameter in the profile of the agent.

3. **Noise Scale**: There is also another hyperparameter that has been used by the BootDqn agent while calculating target output which is known as noise scale. So, the agent stores a specific noise scale value into the replay buffer which is a part of each transition. So, each transition contains a noise scale value for each head of the BootDqn agent. Now, during loss calculation, the noise scale value of the selected head is added to the target or expected output. So, this value is set to 0 by default in the baseline agent. The basic intuition behind tweaking this hyperparameter is the introduction of noise during loss calculation and ultimately making the agent robust to noise and improve the generalization of the agent. Hence, we include this also in the list of experiments to test the influence of noise scale on the agent.

- **A2C**: Till now, we have seen basic value-based agents. In this section, we are using an advanced policy-based agent i.e. A2C agent. Unlike value-based agents where a value function is learnt that would map each state-action pair to a valu [Resb]. We use an on policy method of actor and critic. The actor decides the behaviour of the agent and the critic decides how good is the behaviour of the agent. Hence, we use this agent and

1. **Network Size**: In the default setting the network size of the agent is set to 64 units. The network size contributes to the accuracy of the model. In the literature of the neural networks, the size of the neural network and its use, already states that the deeper neural network has high changes of learning the intermediate features from the inputs provided to the network. The optimal selection of the neural networks has a number of parameters to be considered, which is again out of the scope of our work, hence we only discuss the different network sizes. Here, we have reduced the network size to 32 and doubled the network sizes of the agent. We aim to learn the influence of the network size
on the different Bsuite environments. As we test the agent on a variety of the Bsuite environments each aiming at certain profiling dimensions, we check the influence of the network sizes of the agent.

2. **Max Sequence length:** This parameter restricts the length of the sequence information, stored in the buffer. This information would be used later on to train the agent. We double the value of the maximum sequence length in order to witness its effect on the different core capabilities of the agent. We hypothesize if there is more information in a sequence that is stored and later on used to train the agent, it might have an effect on the memory of the agent.

3. **A2C Hybrid:** Here, we try to create a hybrid A2C agent. The idea behind creating the hybrid agent is to use the best aspects from different agents and to create a hybrid agent. For example, the BootDqn agent uses a prior network and ensembles in its definition, which helps the agent to achieve a good Bsuite score in the exploration dimension, the idea is to combine this ensemble with a definition or architecture of another agent which performs good in another dimension and create a hybrid agent which performs well in most of the dimensions. Contributing to the idea, we tried creating an A2C hybrid agent where we used the best components from A2C and BootDQN agent[Ste19]. As mentioned in the above example BootDqn agent has good exploration scores due to its use of ensembles and prior network. Similarly, A2C agent uses LSTM’s in its core architecture which gives them an upper hand in the memory dimension. We created a hybrid agent where we infused an ensemble of actors in the primary definition of the A2C agent to achieve better exploration at the actor’s end.

In the Figure 3.2, we can the underlying architecture of the A2C hybrid agent. The difference between the basic A2C agent with LSTM’s and our hybrid
agent lies mainly in the actor section, the rest of the architecture remains the same for the two agents. In its basic setting A2C agent, sends the embedding from the LSTM core to the Critic and Actor Linear Layers, where the Critic Linear Layer contains only one unit and the Actor Linear Layer is comprised of the action space of the environment. Here, we replaced the Actor Linear Layer with an Ensemble network and a Prior network which are core concepts used by the BootDqn agent. As shown in the Figure 3.2, the agent accepts inputs(observations), from the environment, process them through the LSTM core to generate embeddings, which are later passed to the critic linear layer to output values and the actor ensemble to output the ensemble of logits. These logits and values are then used to calculate critic and actor losses to train the agent.

- **R2D2:**

  In this set of parameter tunable experiments, we use the R2D2 (Recurrent Relay Distributed DQN) agent [KOQD19]. The R2D2 agent implementation is provided by Acme Framework. We use this hybrid agent to test its capabilities and to test its most influential hyperparameters. The R2D2 agent works on the concept of distributed acting where multiple actors can act in parallel generating experiences and storing them in prioritized experience replay buffer and the learner gets the batch in random to learn the experiences to solve the tasks. Here, we tune the required parameters to see the influence of the hyper-parameters on the R2D2 agent.

  1. **LSTM Size:** We are tuning the LSTM sizes of the agent. The default R2D2 agent uses an LSTM size of 20 cells. We know from the literature that a higher LSTM size can contribute to higher accuracy. The size of the LSTM really depends on the application context and domain. As we are using this agent to profile it and run it over a diverse set of experiments. Therefore, as it runs over a diverse set of experiments we want to test the influence of the LSTM sizes over the range of the Bsuite environments. Hence, we reduce the LSTM to 10 memory cells and double it to 40 cells to check its influence on the R2D2 agent’s profile.

  2. **R2D2 Hybrid:** Here, we are again creating a hybrid R2D2 agent. As mentioned above in the A2C hybrid section, we will use a combination of different components to create a hybrid with good performance scores. The ultimate goal of creating a hybrid agent is to construct a powerful agent that has an overall good profile across all the dimensions. In the default Acme version, the R2D2 uses a simple LSTM network as its actors and learners. We try to use an ensemble of LSTM’s as the base network for the actors and learners in the R2D2 hybrid. We have also implemented unary or binomial masks on top of the ensembles to enable more exploration (The masks and other concepts are described briefly in the BootDqn agent). Hence, we use the underlying concepts from the BootDqn agent which is good in exploration and combine with R2D2 using LSTM core as its network which is good in memory. Hence, our hypothesize is to find out if we create a hybrid agent combing these components can we create an agent who performs well in both exploration and memory tasks.
3.2.2 Bsuite Environments

In this section, we will briefly, look at the different environments provided by the Bsuite [OHA+20]. Bsuite provides a set of 23 diverse environments which are categorized into multiple categories. So, all in all, Bsuite categorizes these environments into 7 major categories namely basic, noise, scale, memory, generalization, exploration, and credit assignment.

Let’s take a look at the different Bsuite environments and their categorization:

- **Basic Learning:** As stated in bsuite[OHA+20] initially they begin with a set of environments to test the basic capabilities of the agent and to test if the agent is competent. This set of environments are targeted to test the agent’s ability to solve a problem, they test the agent’s ability to learn the reward policy. Hence, this set of experiments are classified as 'basic'.

  1. **Simple Bandit:** The environment is a finite armed bandit with deterministic rewards [0, 0.1, ..1] [Git79] with 20 seeds, the interaction are fixed to 10k episodes with scores being normalized regret values where a score of 0 signifies as random and a score of 1 signifies as optimal score. It is only classified in the basic category.

  2. **Mnist:** The environment is a contextual bandit classification of Mnist with +1 or -1 rewards [LBBH98] with 20 seeds, the interaction is fixed to 10k episodes with scores being normalized regret values where a score of 0 signifies as random and a score of 1 signifies as optimal score. It is classified into the basic and generalization categories.

  3. **Catch:** The environment is a 10X5 Tetris grid where one block is falling from the top as per each column. The agent is at the bottom of the Tetris and it can move left or right and has to catch the block. It is again containing 20 seeds, the interaction is fixed to 10k episodes with scores being normalized regret values where a score of 0 signifies as random and a score of 1 signifies as optimal score. It is classified into the basic and credit assignment categories.

  4. **Cartpole:** In this environment, the agent can move right or left on a plane, the motive of the agent is to keep the pole upright moving the cart and hence the name 'Cartpole'[BSA83]. It contains 20 seeds, the interaction is fixed to 10k episodes with scores being normalized regret values where a score of 0 signifies as random and a score of 1 signifies as optimal score. It is classified into basic, generalization, and credit assignment categories.

  5. **Mountain Car:** Here, the agent drives an underpowered car up a mountain [Moo90]. It contains 20 seeds, the interaction is fixed to 10k episodes with scores being normalized regret values where a score of 0 signifies as random and a score of 1 signifies as optimal score. It is also classified into basic, generalization, and credit assignment categories.

- **Stochastic or Noise based Learning:** This section contains the same experiments from basic learning but with the addition of noisy rewards. In this set of experiments, the robustness is being investigated by adding different levels of Gaussian noise. In this set of experiments there are 20 seeds allocated with five
types of Gaussian Noise $N = (0, \sigma^2)$ where $\sigma = [0.1, 0.3, 1, 3, 10]$ with four seeds each[OHA+20].

- **Scale based Learning:** This section contains the same experiments from the basic learning but under the different scales of rewards. In this set of experiments, the robustness of RL agents is being investigated by multiplying rewards different levels of $\lambda$. In this set of experiments there are 20 seeds allocated where the rewards is multiplied by five levels of $\lambda = [0.01, 0.1, 1, 10, 100]$ with four seeds each[OHA+20].

- **Exploration:** The RL agent interacts with the environment, it does not have the complete picture of the environment. The agent learns the environment with the help of its interaction with the environment. During the interaction, the agent uses the knowledge of previous states and rewards to take an action. Now, the dilemma in front of the agent is to explore the poorly known states and select action based on the strategy of good future rewards or exploit the poorly known states to get a good short performance. This exploration vs the exploitation dilemma is a challenging problem as exploration is an essential core capability of the agent. The problem of exploration deals with prioritizing the useful information for learning and this set of environments are designed to investigate the deep exploration [OVRR+19] capabilities of the RL agent.

1. **Deep Sea:** As shown in the Figure 3.3, In this experiment, there is a $N \times N$ Grid. Here, all the agent starts from the top left corner, where at each time step it can take either a left or right and at each time step it would gradually be shifted a row below it. The game ends when the agent completes the $N$ step or reaches the bottom-most row. Now, the catch in this experiment is the rewards agent gets on each transition or action selected. If the agent selects a left action it gets no cost i.e. $r = 0$ and if it takes the right action there is a small cost of $r = 0.01/N$. The bottom right corner contains the treasure and on reaching that state the agent gets a reward of $+1$. Hence, the agent has to be fundamentally designed to explore in the long term and it should be willing to ignore the small costs incurred at the early stage in order to maximize the cumulative rewards[OHA+20].

It has a deep-sea environment with $N = [5..50]$, interaction size of 10k episodes

![Figure 3.3: Deep Sea Exploration](image)
with scores being % of runs with average regrets less than 90%. This experiment is only classified in the exploration category.

2. **Stochastic Deep Sea:** This environment is the same as deep-sea but the transition is stochastic in nature i.e., it contains noise rewards in its environment. This experiment comprises deep-sea chain environments with stochastic transitions. It has an interaction of fixed 10k episodes with scores with average regrets as the deep sea. This set of experiments are classified in exploration and stochasticity or noise-based learning\([OHA+20]\) categories.

3. **Cartpole Swing-Up:** The Cartpole Swing up problem \([BSA83]\) is similar to the 'Cartpole' environment but with sparse rewards. It has a height limit from 0 to 0.95. Unlike the other environments, the interaction here isn’t fixed to 10k episodes but it is fixed to 1k with scores being % of runs with average regrets greater than 0. This experiment is classified into exploration and generalization categories.

- **Credit assignment:** In this section we will briefly go through the credit assignment environments. The RL algorithms have extended their contextual bandit problem to allow long sequences. They permit the entry of long-term sequences in order to strengthen the concept of sequential decision making \([OHA+20]\). This implies that the actions selected in an earlier time step have the capacity to affect the performance in the later time steps. This problem set is nothing but credit assignment and this set of experiments are targeted to investigate the same in modern-day RL algorithms.

1. **Umbrella Length:** This experiment was selected because of its capacity to evaluate the credit assignment problem. It is called as stylized 'umbrella problem' \([OVRR+19]\). The experiment is to carry an umbrella at the start of the problem and then there would be a step on unrelated tasks and finally rain. The confounding factor in these long chains of unrelated would vary in length, logarithmically from 1 to 100. It has an interaction fixed to 1K episode with scores with normalized regrets where a score of 0 signifies as random and a score of 1 signifies an optimal score. It is classified under credit assignment and stochastic categories.

2. **Umbrella Features/Distract:** This experiment is similar to umbrella length environment\([OVRR+19]\). The experiment is to carry an umbrella at the start of the problem and then there would be a step on unrelated tasks and finally rain. The confounding factor in these long chains of unrelated would vary in features, logarithmically from 1 to 100. It has an interaction fixed to 1K episode with scores with normalized regrets where a score of 0 signifies as random and a score of 1 signifies an optimal score. It is also classified under credit assignment and stochastic categories.

3. **Discounting Chain:** This is the last experiment under the credit assignment category. The description of the experiment is that it is related to discounted horizon problems. It is specifically designed to highlight the discounted horizon with 1k episodes as an interaction. Its scores are comprised of normalized regrets where a score of 0 signifies as random and a score of 1 signifies as optimal score and categorized into only credit assignment category.
• **Memory**: Memory is always been an important aspect of being competitive to complete certain tasks. Almost every agent uses some form of memory to solve a problem. The agent’s ability to remember the state representation in a long series of observations to solve tasks is the main aspect that is being investigated by this set of experiments. The agents possessing the ability to know the historical state representation help them to solve the tasks very effectively. Hence, we will take a brief look into the experiments related to memory in this section

1. **Memory Length**: This experiment is based on a T-maze experiment where the rats have to use their cognitive abilities about the spatial location to solve the T maze problem [OD71]. Here, the environment contains T-maze with a single binary context which grows logarithmically from 1 to 100[OHA+20]. It has an interaction fixed to 1K episode with scores with normalized regrets where a score of 0 signifies as random and a score of 1 signifies an optimal score. It is classified under credit assignment and memory categories.

2. **Memory Bits**: In this experiment, the environment contains T-maze with length 2. Here, the agent has to learn a varying number of bits to remember. The number of bits in this experiment grows logarithmically from 1 to 100[OHA+20]. It has an interaction fixed to 1K episode with scores with normalized regrets where a score of 0 signifies as random and a score of 1 signifies an optimal score. It is classified under credit assignment and memory categories.

This brings up to the end of the Chapter 3, where we initially saw the research questions which we are targeting to scientifically address in our work. After the research questions, we saw the design of the profiling of agents in detail. In this section, we mainly dug deeper into the subsection viz. Tunable Parameter where we saw the different hyperparameters related to the modern-day DRL algorithms. Here, we also looked at the importance of the hyperparameters and our hypothesis about why we have selected this respective set of hyperparameters in our work. Then we switched to the Bsuite provided environment. In this subsection, we discussed the different Bsuite environments in detail. In the Bsuite environments section, we also discussed the different categories of the environments and the different sets of experiments under their respective category.
4 Experimental Set-Up

In the previous chapter Chapter 3 we saw the research questions and the design of the profiling of agents. We went through the different tunable hyperparameters, their definitions and their influences on their respective DRL algorithms. We also brushed through the different bsuite environments and their descriptions along with the set of their interactions, reward system and their categorizations into different dimensions or core capabilities of RL algorithms.

In this chapter, we are going to look at all the necessary components used in our work, their software specification and the required configuration for reproducing our work and findings. The Chapter 4 is structured as follows:

The chapter is divided majorly into two sections viz. the DRL frameworks used in this work and the Code Configuration. In the first section we have mentioned in brief the two DRL frameworks we used in our work i.e. 'Bsuite'[OHA+20] and 'Acme'[HSA+20], the underlying purpose of the frameworks and describe in detail the services provided by the DRL frameworks. In the second section, we discuss the different tools, software, versions and configurations that we have used in our work. We present all these details in order to reproduce the results and carry future research in this domain.

4.1 DRL Frameworks:

Here, we discuss the two major DRL frameworks viz. 'Bsuite'[OHA+20] and 'Acme'[HSA+20]. In this section, we also discuss the fundamental uses of the frameworks and what components of the frameworks we have used in our work. So let’s begin with the following DRL frameworks.

4.1.1 Bsuite:

During the beginning period of research on DRL, the RL algorithms have been grounded to their performances in specific environments. The research and the development of the RL algorithms and DRL algorithms were evaluated based on their performances in environments like 'Cartpole'[BSA83] and 'Mountain Car'[Moo90]. In the literature, very little research was made in the evaluation of RL agents based on their core capabilities. Bsuite is a framework that addresses this problem. Bsuite is a benchmarking framework, which aims to benchmark the modern DRL agents based on their behaviour. It employs a specific methodology to evaluate the agent beyond certain specific evaluation environments. Bsuite evaluates the DRL agent based on its core capabilities which are not only restricted to the agent’s performance on certain environments like 'Mountain Car'[Moo90] and 'Cartpole'[BSA83]. It isolates the core capabilities of the agents and
test the isolated capabilities like scalability, exploration etc, rather than testing the general learning ability of the agent. For the benchmarking of the agents, it has a set of meticulously designed experiments which we refer to as bsuite environments. These experiments are also added based on their five key qualities (Simple, Challenging, Scalable, Targeted and Fast) [OHA+20]. Bsuite provides a bsuite score which in turn can be used to create plots like radar plots. The bsuite scores are typically based on various measures like average regrets and normalized average regrets. The bsuite scores are also generated average regrets per 10k or 1K episodes depending upon the environment the agent is being evaluated against. Running the agent over the entire set of 23 environments would provide us with the bsuite score of agents over this entire set of environments. We can plot a histogram of bsuite scores vs the experiments to get an intuition about the agent’s profile in depth. As mentioned above we can also generate radar plots, where the bsuite aggregates its environment scores into the 7 categories of environments. Hence, providing the agent’s profile over different dimensions, where we can see the agent’s scores in each of the dimensions. This helps us to benchmark various agents and get an intuition of the agent’s or RL algorithms performance over different dimensions. They not only aim to provide a score about the agent performance through the different, but they also provide additional services like logging of the result, generate histograms and provide more informative value to the evaluation of the agents. Hence, their research behind bsuite derives its motivation from the development of powerful RL algorithms. With the help of bsuite, the researchers can develop various RL algorithms as researchers now can now visualize the RL algorithms performance on different dimensions and they can use this knowledge to further develop more powerful RL agents.

4.1.2 Acme:

There have enormous amounts of advancements in the field of development of modern RL algorithms. With the development of modern-day RL algorithms, there is a comprise made on the scalability and complexity of the RL algorithms. The modern-day RL algorithms have been more and more complex in order to achieve better performance. The increasing complexity of the advanced RL algorithms has even made it difficult for researchers to reproduce the RL algorithms. To resolve the above problem and to facilitate reproducibility of the RL algorithms, acme[HSA+20] comes into the picture. Acme[HSA+20] is a DRL framework that targets the issue of RL algorithms being more complex and provides a base implementation of various RL algorithms that the researchers can use to develop RL algorithms that are easily reproducible. Acme provides a simple implementation of various RL algorithms which are experimented on various levels of complexity and computation. They aim to deliver such agent implementations in their framework such that the results of modern-day developed algorithms are reproducible by the researchers. They provide a simple implementation for a variety of RL algorithms that could be scaled up and down and it can achieve great parallelization too.

Acme is a software implementation and a lightweight DRL framework that facilitates the development of RL algorithms to address the scalability and complexity issues within a single framework[HSA+20]. They provide components at different levels of implementation, from the very granular level like the implementation of rewards, policies, networks, etc to the higher abstract level implementations like actors, learners, buffers etc. It encourages the researchers to not only develop scalable and robust RL algorithms but also
evaluation tools to evaluate the new agents. It has compatibility and various evaluation measures for the researchers to choose from. It also incorporates a bsuite framework in order to evaluate the compatible agents. We use this feature of the acme framework to store the performances of the agent and use the bsuite framework to get the profile of the advanced agents.

**Algorithm 4:** Code factoring to change the traditional environment loop in acme. ([HSA+20])

```plaintext
while True do
    Make initial observations
    Reset the environment to $O_0$
    Observe $O_0$
    while not last step do
        Evaluate the policy and take a step in the environment.
        Select an action and take a step
        Make an observation and update the actor.
        actor observes action and it updates itself
    end
end
```

**Figure 4.1:** Environment loop for interaction between actors and environment [HSA+20]
As discussed above, acme aims to provide a simple description of the RL algorithm so that it could be tested at different levels of scalability. To achieve the various levels of scalability and parallelization, it supports the distributive RL agents as well. So it supports the concept of multiple actors interacting with the environment in a distributive setting as well as a simple actor interacting with the environment all in a unified framework. This is one of the key features of the acme framework. As shown in the Figure 4.1 and the pseudocode in the line 1, we can see the expansion of the environment interaction loop in the acme framework. It makes use of components of reverb table and replays buffer in order to store experiences from the multiple actors and then learners using this buffer to learn and update its policies. The acme framework operates in both the settings from a simple agent environment interaction to the distributed RL agent, with the help of the same building blocks. The same exact modules are used in the acme framework with a limited number of changes and they achieve this by factoring their components code as shown in the line 1.

As shown in the Figure 4.1, we can see that acme has made changes to the internals of the agent and they term this agent as a learning agent. It is called a learning agent as it has two major components in the agent i.e. actors and learners. The actors perform the selection of the actions as per their policy and store the experiences in the dataset. The experiences are then passed to the learner which gets the experiences in mini-batches learns how good is the policy of the actor and updates the policy of the actors by providing feedback and update its own network to improve its feedback quality. Hence, they keep this underlying strategy for the development of various agents and evaluate them to reduce issues in complexity and scalability.

4.2 Softwares, Versions and Code Configurations:

In this section, we will go through the different software used in our work, their corresponding versions, the system and the server details on which the experiments were executed. In the next subsection, we would discuss in brief the code structure and the code changes we did in order to run the respective experiments related to the research questions, which are mentioned in the Chapter 3.

4.2.1 Softwares and Versions:

In our work, we have used the bsuite library and the acme framework. We have used bsuite as a benchmarking tool where we use the baseline TensorFlow implementations of the agents provided by the bsuite. We then make changes in the hyperparameters of the bsuite agent based on the tunable parameters and experiments list Subsection 3.2.1 that is mentioned in the Chapter 3. All the aforementioned agents in the Chapter 2 are either a baseline implementation from either Bsuite or Acme. Most of the agents used in our work like DQN, A2C and BootDQN are baseline implementations from Bsuite whereas hybrid agent R2D2 has been the only baseline agent from Acme. All the experiments performed and the report generated in this report have used bsuite reporting components. Even when we used the Acme framework, we used bsuite logging mechanisms to log the experiment
results in .csv files which we would later use to generate different graphs like radar plots and histograms to give the profile of the RL algorithms.

We have used Colab Notebooks and Jupyter Notebooks for all the experiments we ran in our thesis. During the early phases of the project when we were heavily invested in thinking about the design and hypothesis of the different experiments we used Google Colab to run the Bsuite agents to get familiar with the different hyperparameters. During the later half of this project, we used Jupyter Notebooks to execute the experiments. All the experiments at the Google Colab Notebooks were run through a Normal Laptop with average computing power and it had a configuration of 4GB RAM, 512 GB Hard Disk, Intel i3 gen chipset. As it was on Google Colab the configurations of the system did not have much effect on the speed and performance of experiments. We used the default GPU provided by Google Colab. The latter half of the experiments that have performed on the Linux Server had a configuration of x86_64 Linux 2.6.32.59, with 32 Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz processors and 256 GB of RAM. All the experiments and reports being generated and used in this report are obtained using the bsuite v 0.3.5 and acme v 0.2.2.

4.2.2 Code Structure and Configurations:

As mentioned above we have used Jupyter and Colab Notebooks for all the experiments. We started by referring to the bit.ly/bsuite-tutorial notebook provided by bsuite\cite{OHA+20} in order to get familiar with the bsuite environment and agents. Similarly, the acme framework too had the tutorial notebook about how to install and use the agents provided by the acme framework \cite{HSA+20}. The instructions about the installation and configurations are provided in the GitHub of both the frameworks used in our work. We also used the same instructions for installing and configuring the frameworks so that we can use the baseline agents and run our list of experiments.

As already mentioned by bsuite \cite{OHA+20} in their work about the ideal guidelines for creating an agent. The recommended method suggests creating a class for the agent which comprises of a policy method to take the actions, an update method to learn from the experiences, the transitions and rewards. Finally, passing the agent to the environment loop which executes the agent on the entire set of experiments provided by bsuite in the Subsection 3.2.2. Finally, save the results into the local system and then you can use these files to generate plots and reports for your analysis. One can customize the environment loop to also run only a limited set of environments using the names of the environments. Also, bsuite environments have different configurations, if we use ‘sweep.SWEEP’ in the environment loop, which contains all the seed configurations of all the experiments or in case the agent’s performance needs to be tested on all seed configurations of some specific environment we can mention ‘sweep.environments name’. For example, if we use a sweep.CARTPOLE in the environment loop all the seeds starting from ‘cartpole/0’ to ‘cartpole/20’ would be executed. We used the bsuite agent and use to make changes in the underlying action method, networks, replay buffer and the set of hyperparameters used in the implementations of the baseline agents provided by the bsuite for our research. (During changing various policies, rewards and networks we faced a couple of roadblocks due to the TensorFlow implementation of the agent being in the TensorFlow 1.0 version where many of the functions are updated and removed in the TensorFlow version 2.0. We
used TensorFlow 2.0 constructs and hence changed the functions with their alternatives in the TensorFlow 2.0). After making the changes in the underlying baseline TensorFlow based agents provided by the bsuite, we used the same environment loop constructs to run the agent and generate results.

For the acme framework[HSA+20], we use the same strategy of cloning the framework using their git and make changes in the underlying code structure and the RL algorithm. Unlike bsuite, the acme does contain all the policies in a single agent class rather it has its own nuances. The structure of the baseline agents provided by acme consists of actor, learner architecture. So we had to make changes majorly in 4 files namely: agent.py, actors. py, learning.py and savers.py. The agent contains the method calls actors and learners. The actors and learners contained policies and networks required for their operation. Acme framework also contained a special buffer table called a reverb table to store experiences from multiple actors in a distributed setting. The savers.py contained all the necessary code about what to store in the buffer and the buffer limit. Therefore, we had to make changes in these four files at the acme framework before executing the agent at the Colab Notebook. In the notebook, we used a class to define the agent, where researchers can add all the initial observation and state values for the agent in different methods and finally use the environment loop to pass the agent and the environment’s names to execute the agent on the given environment and generate results. They also provide their logging and reporting mechanism default to their framework but we have used bsuite logging even to log acme agents on bsuite environments. We have used an R2D2 agent from the acme framework and created a hybrid agent using all these building blocks in our work.

This brings us to the end of the Chapter 4 where we discussed the different DRL frameworks like bsuite and acme in depth. We explained the frameworks in-depth, explaining the underlying architecture of these frameworks. Next, we saw the different Softwares like Google Colab and Jupyter Notebooks along with their versioning and system requirements we used to run the experiments. Finally, we saw the code structure and the configurations we did in the agent of the frameworks to build new agents and run our experiments.
5 Evaluation and Results

Till now, we have discussed the background of deep reinforcement learning, its challenges, the different families of DRL agents, their designs, architectures, hyper-parameters and evaluation matrix. In this chapter, we will take a look into the evaluation and discuss in brief about the different set of experiments that contributes cohesively to our research questions discussed in chapter 3. We will also analyse the results of each experiment and provide a tabular summary of the analysis at the end of this chapter.

This chapter is divided among the 3 research questions mentioned in chapter 3. Now, each research question is again broken down into a set of experiments, where we discuss the setup of the experiment, the hypothesis of the experiment, the results of the experiments and the discussion section. In the discussion section which is the final block of every experiment we will be validating our hypothesis. Furthermore, for all the given experiments we used basic agent definitions provided by bsuite and acme and we have either made changes to their hyper-parameters or made changes to their architectures in order to test our hypothesis.

Note: There is an important note that needs to be mentioned before starting with the experiments. Each of the environments provided by bsuite which we have used to further evaluate our agents and validate our hypothesis contains around 20 configurations for the results to be reproducible. Initially, we checked into the configurations and we found that the only thing variable in each of the configurations is the speed with which the environment begins and the results are reproducible for the initial environments. Hence to save time and run a variety of hypotheses, we decided to run only one configuration for our analysis. At the very later end, when most of our experiments had been executed we realised that the results for many of the complex environments are not reproducible and hence we had to redo all of the experiments we had executed. Due to resource and time limitations, we could only run a selective set of experiments for the total number of configurations provided by bsuite. Nevertheless, we decided to include the results and the findings of the experiments we ran for only 1 configuration in order to share our findings and in case anyone wants to further investigate any findings can run 20 configurations and validate their hypothesis.
5.1 Experiments: Research Question 1 (Hyper-parameter Changes)

This section contains all the experiments which were aimed to find the influence of hyper-parameters on DRL agent implementations provided by bsuite and acme frameworks.

Under this section, we have used the most common agent i.e. DQN agent and we have changed different hyper-parameters of the agent and observed the behaviour of DQN to all these changes. Apart from DQN, we have also used different basic and advanced agents such as BootDqn, A2C and R2D2 where DQN, BootDqn and A2c agent implementations are provided by bsuite and R2D2 agents implementation is provided by acme.

5.1.1 DQN Agent

Due to the configuration issues faced we did not run the default DQN agent for all 20 configurations but with 5 configurations it shows the same behaviour. So, we have used the results of 5 configurations as the default DQN for all the below experiments in this section.

Learning Rate:

- Set-Up: In this experiment, we are tuning the values of the hyper-parameter learning rate of the DQN agent provided by bsuite. The learning rate value used by the
Figure 5.2: Radar plot: Learning Rate of default DQN (0.05) vs 0.01 vs 0.1

Figure 5.3: Histogram: Learning Rate of default DQN (0.05) vs 0.01 vs 0.1
default DQN is 0.05. We reduced the learning rate value to 0.01 and doubled the learning rate value from 0.05 to 0.1 and ran the DQN agent over the set of 23 experiments but for only 1 configuration.

- **Hypothesis:** When we increase the learning rate in deep neural networks the network quickly converges and when the learning rate is reduced the network converges slowly towards the global optimum. Also, on increasing the learning rate the agent can explode and not converge at all. Hence, the hypothesis of this particular experiment is inspired by the above-mentioned behaviour of learning rates. Therefore, we increased as well as decreased the learning in order to test the changes in the agent w.r.t to the exploration dimension.

- **Discussion:** In Figure 5.2 we have a radar plot showing us the values of DQN agents with different learning rates (0.05, 0.1 and 0.01). Surprisingly, we can see that the agents with learning rates of 0.1 and 0.01 are better in terms of memory than the default agent. Rest in all the other dimensions the default DQN with learning rate performs better. The agent with a learning rate of 0.1 performs poorly in almost all the dimensions as compared to others. The one with lr = 0.01 performs better than the DQN agent with lr = 0.1 but not better than lr = 0.05 except memory.

To further investigate the results, we generated a histogram of the results of the different DQN agents for the overall set of experiments. In Figure 5.3 we can see that the DQN agents with lr value 0.01 and 0.1 have a bsuite score of 1.0 for memory length and hence the spike in the memory dimension. Hence, the finding of this experiment is hyper tuning the learning has the potential to increase the memory of the agent. As we ran the agent for only one configuration, future work needs to be done in this direction to validate our findings and how influential the learning rate hyperparameter could be in terms of the memory of the agent.

**Epsilon:**

- **Set-Up:** In this experiment, we are tuning the values of the hyper-parameter Epsilon of the DQN agent provided by bsuite. The epsilon value used by the default DQN is 0.05. We have reduced the epsilon value to 0.025 and doubled the epsilon value from 0.05 to 0.1 and ran the DQN agent over the set of 23 experiments for 20 configurations.

- **Hypothesis:** The epsilon hyperparameter decides the randomness of the action taken by the agent i.e. if the epsilon is high the agent explores more and decides to choose a random action than exploit the best action and if the epsilon is low the agent does not explores more and exploits the best option. Hence, the epsilon parameter maintains a balance between exploration and exploitation. Therefore, we have increased and decreased the epsilon parameter to test its influence on the exploration dimension of the DQN agent.

- **Discussion:** In Figure 5.4 we have a radar plot showing us the values of DQN agents with different epsilon values (0.05, 0.025 and 0.1). In Figure 5.4, we can see that instead of increasing or decreasing the epsilon parameter, there is no change in the exploration dimension of the agent. Instead, the credit assignment and the noise
Figure 5.4: Radarplot: Epsilon of default DQN (0.05) vs 0.025 vs 0.1

dimension of the agent has been reduced w.r.t the default DQN. We can also see that the performance for all the dimensions is almost the same for epsilon values of 0.025 and 0.1. From the results, we can state that the exploration of the DQN agent remains unaffected by changes in the epsilon parameter, rather few dimensions like noise and credit assignment are influenced by the same.

Boltzmann DQN:

- **Set-Up:** In this experiment, we attempt to use the Boltzmann equation in order to increase the exploration of the DQN agent. The default DQN uses the epsilon greedy strategy to select its action and we decided to change the strategy to the Boltzmann equation in order to increase the exploration of the agent. Here, we are running Boltzmann DQN (i.e. DQN with Boltzmann policy) with different values of epsilon parameter to test its exploration capacity. We run the Boltzmann DQN agent over the set of 23 experiments for 20 configurations.

- **Hypothesis:** The default DQN uses Epsilon Greedy policy as already mentioned above, it chooses a random action and ignore the best action until a certain minimum value of epsilon hyper-parameter. After, the minimum value is reached it starts choosing the optimal value and hence exploitation of the optimal actions. Here, we used the Boltzmann equation instead of Epsilon greedy as we no longer choose the optimal action, we normalize the Q values and then choose the action randomly in order to explore the exploration.

- **Discussion:** In Figure 5.5 we have a radar plot showing us the values of Epsilon Greedy DQN vs Boltzmann DQN with epsilon values of 0 and 0.05). In Figure 5.5, we can see that irrespective of changing the policy to Boltzmann and changing the epsilon parameter, there is no change in the exploration dimension of the agent. Instead, the credit assignment and the noise dimension of the agent has been reduced.
Figure 5.5: Radarplot: Epsilon Greedy DQN vs Boltzmann DQN with Epsilon 0 and 0.05 w.r.t the default DQN. We can also see that the performance for all the dimensions is almost the same for both the epsilon values of Boltzmann DQN. From the results, we can state that the exploration of the DQN agent remains unaffected by changes in the action selection policy from Epsilon Greedy to Boltzmann Equation, rather few dimensions like noise and credit assignment are influenced by the change.

Optimizer:

- **Set-Up:** This is the final experiment with DQN related to research question 1. So, in this experiment, we are changing the optimizer of the agent. The default DQN uses Adam Optimizer and in this experiment, we change it to Rmsprop Optimizer. We run the agent for only 1 configuration and compare it with the Default DQN.

- **Hypothesis:** The default DQN uses Adam Optimizer, we change the optimizer to Rmsprop in an attempt to exploit the exploration dimension.

- **Discussion:** In Figure 5.6 we have a radar plot showing us the values of DQN with Adam optimizer and Rmsprop optimizer. In Figure 5.6, we can see that the exploration dimension still remains unmoved irrespective of the optimizer being used but we can see a noticeable difference in other dimensions. We can see that DQN with Rmsprop optimizer performs poorly in terms of Scale and Basic Dimensions but it has significantly improved the memory dimension.

To further investigate the results, we generated a histogram of the results of the agents for the overall set of experiments. In Figure 5.7 memory dimension. Hence, the finding of this experiment is that usage of different optimizer have the potential to increase the memory of the agent. As we ran the agent for only one configuration, future work needs to be done in this direction to validate our findings and how influential the optimizer of an agent could be in terms of memory dimension.
Figure 5.6: Radarplot: Default DQN (Adam Optimizer) vs DQN (Rmsprop Optimizer)

Figure 5.7: Histogram: Default DQN (Adam Optimizer) vs DQN (Rmsprop Optimizer)
5.1.2 BootDqn Agent

Till now we saw the different experiments related to the DQN agent. In this subsection, we will look at all the experiments related to the BootDqn agent.

Due to the configuration issues faced we did not run the default BootDqn agent for the overall set of 20 configurations. Hence, we will refer to the BootDqn agent Figure 5.1 from all the bsuite agent profiles mentioned at the start of this chapter. We can see from the image that BootDqn has a very good bsuite score in exploration, generalization dimensions. It has a score of almost 0.5 or more in all the dimensions except the memory. Hence, in this section, we will have a look at different hyper-parameter experiments related to memory dimension.
**Mask:**

- **Set-Up:** In this experiment, we tune the mask of the agent. The default definition of the BootDqn agent consists of a binomial mask (i.e., mask with one’s and zero’s) for different heads of the ensemble. We change this binomial mask into a unary mask with all one’s, so that all the heads are considered. Here, we run the agent for only 1 configuration and compare it with the Default agent.

- **Hypothesis:** The default BootDqn uses an ensemble where each network in the ensemble is referred to as a head of BootDqn agent. Now, there is a binomial mask applied on top of the heads with values 0 and 1. This mask filters the heads where the value of the mask is 0 and considers only the heads where the mask value is 1, which helps in enabling a good exploration score. We change this mask with all one’s so that all of the heads are considered and run the experiment to check if the exploration is really affected by this change.

- **Discussion:** In Figure 5.8 we have a radar plot showing us the values of BootDqn with mask all one’s i.e. $m_t=1$ instead of $m_t=0.5$. In Figure 5.8, we can see that the exploration and the generalization dimensions have been reduced w.r.t to the default BootDqn as shown in Figure 5.1. We can also notice changes in the values of other dimensions. The memory, credit assignment and noise dimensions show a noticeable increase in their bsuite scores.

Furthermore, to investigate the results, we generated a histogram of the agent’s results for the overall set of experiments. In Figure 5.9 we can see that the BootDqn agent with $m_t=1$ we can see the bsuite score of 1.0 for memory length, umbrella length, distract and some experiments under the noise category. Hence, we can see a rise in the bsuite scores of the respective dimensions. As we ran the agent for only one configuration, future work needs to be done in this direction to validate our findings and how influential masks of the BootDqn agent could be in terms of memory, noise and credit assignment dimensions.

**Ensemble Heads:**

- **Set-Up:** In this experiment we are tuning the number of ensembles (heads of the agent). The default agent consists of 20 networks or heads in the ensemble. Hence, we tune the no. of agent’s heads in this experiment. We run this experiment with 10 and 30 no. of heads for all 20 configurations. We also run another experiment in this section with 40 heads and only configuration.

- **Hypothesis:** The default agent consists of 20 heads in its ensemble. BootDqn has a good exploration and generalization score and we suspect that the agent’s use of ensemble in its definition is an influential factor for the score. Hence, we change the number of heads to 10, 30 and even 40 to check its influence on exploration and memory.

- **Discussion:** In Figure 5.10 we have a radar plot showing us the values of BootDqn with 10 and 30 heads. In Figure 5.10, we can see that both the agents have almost the same values among all the dimensions. As compared to the default BootDqn
Figure 5.10: Radar Plot: BootDqn agent with 10 and 30 Ensemble heads

Figure 5.11: Radar Plot: BootDqn agent with 40 Ensemble heads
agent Figure 5.1 we can clearly observe that the exploration and generalization have been reduced without significant improvements in any dimension.

Furthermore, we also had results from an earlier experiment in the section. In the given experiment, we had increased the value of the heads to 40 and reran the agent. Due to time constraints, we could rerun the same for 20 configurations, but we generated a radar plot to share our findings. In Figure 5.11 we can see that the BootDqn has improved its bsuite score in terms of memory, noise and credit assignment dimensions. we ran the agent for only one configuration, future work needs to be done in this direction to validate our findings and how influential 40 heads of the BootDqn agent could be in terms of memory, noise and credit assignment dimensions.

**Noise Scale**

- **Set-Up:** This is the final experiment with BootDqn related to research question 1. So, in this experiment, we are tuning the noise scale of the agent. The default BootDqn uses a noise scale of 0, we increase the noise scale of the agent to 0.5 and 1. We run the agent for only 1 configuration and compare it with the Default BootDqn agent.

- **Hypothesis:** DRL agents needs to be effective against uncertainty. To be effective against uncertainty and to generalize better noise values are added to the rewards values of the agent. In the default setting, the noise scale value is set to 0. In this experiment, we increase the value to 0.5 and 1 in order to check its influence on Generalization, Noise and other dimensions.

- **Discussion:** In Figure 5.12 we have a radar plot showing us the BootDqn agent with noise scale values of 0.5 and 1. In Figure 5.12, we can see that the generalization
and exploration dimensions have reduced and there is a little increase in the score of memory dimension. As we ran the agent for only one configuration, future work needs to be done in this direction to validate our findings and how influential the noise scale of an agent is to the other dimensions.

### 5.1.3 A2C Agent

Till now we saw the different experiments related to the BootDqn agent. In this subsection, we will look at all the experiments related to the A2C agent which is the last bsuite agent in this category.

Due to the configuration issues faced we did not run the default A2C agent for the overall set of 20 configurations. Hence, we will refer to the A2C agent Figure 5.1 from all the bsuite agent profiles mentioned at the start of this chapter. We can see from the image that the BootDqn has a decent bsuite score in basic, generalization and memory dimensions. It performs poorly in terms of the exploration dimension. In this section, we will only tune max seq length in an attempt to check its influence on other dimensions.

#### Max Seq Length

- **Set-Up:** In this experiment, we tune the max seq length hyper-parameter by doubling its value to 64 from 32 which is its default setting. Here, we run the agent for only 1 configuration and compare it with the Default agent.
- **Hypothesis:** The default A2C has a maximum sequence length value of 32. We have increased the value to 64 to store more sequences in the buffer. In this experiment, we are aiming to test if the memory dimension would be affected by changing the value of seq length.

- **Discussion:** In Figure 5.13 we have a radar plot showing us the values of an A2C agent with seq length set to 64. In Figure 5.13, we can see that the memory dimension remains unchanged. The bsuite scores of exploration and generalization also sank but we could see a spike in the credit assignment and noise dimension. As we ran the agent for only one configuration, future work needs to be done in this direction to validate our findings and how influential the max seq length of the A2C agent could be in terms of other dimensions.

### 5.1.4 R2D2 Agent

In the previous sections, we ran experiments on bsuite provided baseline agents. In this section, we will run an experiment on the agent provided by Acme. Hence, in this section are going to run experiments on the R2D2 agent. Also, this is the final subsection in this block of Research questions 1.

Due to the configuration issues faced we did not run the default A2C agent for the overall set of 20 configurations but we managed to run the agent for 1 configuration. As, it is not a bsuite provided agent we do not have a profile of the default agent, so for references purposes, we are using the default agent with 1 configuration as our default R2D2 agent in this setting.
LSTM size:

- **Set-Up:** In this experiment, we tune the size of the LSTM used in the agent. The default R2D2 uses LSTM of size 20, we increased it to 40 and reduced it to 10. Here, we run the agent for 20 configurations and compare it with the Default R2D2 agent (1 configuration).

- **Hypothesis:** The default R2D2 agent has an LSTM size of 20. The more the size of LSTM, the better the agent learns about the input data. The optimal size of the LSTM depends on the complexity of the input data. As we are using the bsuite set of experiments for profiling the agent, the input data would vary from experiment to experiment. Hence, we aim to increase and decrease the size of the LSTM and check if there is any impact on the profiling dimensions of the bsuite.

- **Discussion:** In Figure 5.14 we have a radar plot showing us the R2D2 agents with different LSTM sizes of 10, 20 and 40. In Figure 5.14, we can see that none of the profiling dimensions shows an improvement w.r.t to the default agent. Instead, it performs poorly on all the dimensions except memory, scale and basic category of experiments. We can also see that the profiles of the R2D2 agents with LSTM sizes of 10 and 40 are almost the same. As we have used the default R2D2 agent and ran it for only 1 configuration, more research needs to be done on this topic to understand the behaviour of the agent and the influence of LSTM sizes on them.

### 5.1.5 Summary of RQ1 Exp’s:

Till now, we have seen all the experiments related to the first research questions. We began the list with bsuite agents like DQN, BootDqn, A2C and ended the list with R2D2 agent of Acme framework. We further divided the agents in terms of the experiments done on the agents. In each of the experiments we saw the set up of the experiment, hypothesis and finally a discussion section based on the results of the experiments. In this section, we are doing nothing new, we are just summarizing what we have seen in this section in the form of the table Table 5.1.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Framework</th>
<th>Hyper-parameter Tuned</th>
<th>Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>Bsuite</td>
<td>Learning Rate</td>
<td>Improves memory</td>
</tr>
<tr>
<td>DQN</td>
<td>Bsuite</td>
<td>Epsilon</td>
<td>Performs poorly in noise and credit</td>
</tr>
<tr>
<td>DQN</td>
<td>Bsuite</td>
<td>Boltzmann</td>
<td>Performs poorly in and credit</td>
</tr>
<tr>
<td>DQN</td>
<td>Bsute</td>
<td>Optimizer</td>
<td>Improves memory</td>
</tr>
<tr>
<td>BootDqn</td>
<td>Bsute</td>
<td>Mask</td>
<td>Improves memory, noise and credit assignment</td>
</tr>
<tr>
<td>BootDqn</td>
<td>Bsute</td>
<td>Ensemble Heads</td>
<td>Only 40 heads performs better in memory, noise and credit assignment</td>
</tr>
<tr>
<td>BootDqn</td>
<td>Bsute</td>
<td>Noise scale</td>
<td>Improves memory dimension.</td>
</tr>
<tr>
<td>A2C</td>
<td>Bsute</td>
<td>Max Seq Length</td>
<td>Improves noise and credit assignment</td>
</tr>
<tr>
<td>R2D2</td>
<td>Acme</td>
<td>LSTM size</td>
<td>Performs poorly in all dimensions</td>
</tr>
</tbody>
</table>

**Table 5.1:** Summary of RQ1 Exp’s
5.2 Experiments: Research Question 2 (Architecture Changes)

This section contains all the experiments which were aimed to find the influence of architecture changes on various RL algorithm implementations provided by mostly bsuite frameworks.

Under this section we have used the most common agent i.e. DQN agent and we have made changes in the underlying network of the agent and observed the behaviour of DQN to all this changes. Apart from DQN , we have also used advanced agents such as A2C agent. We made similar changes in the network of the A2C agent implementations and observed the behaviour of the agent w.r.t the entire set of environments offered by bsuite. In this section we make changes in the architectures of RL algorithms to observe the impact on their performance. We also present our findings in this section to highlight the influence of the architectural changes in DRL algorithms.

5.2.1 DQN Agent

Again, due to the configuration issues faced we did not run the default DQN agent for all 20 configurations but with 5 configurations it shows the same behaviour. So, we have used the results of 5 configurations as the default DQN for all the below experiments in this section.

Network Drop out:

- **Set-Up:** In this experiment, We make use of the drop out function to drop out 20%, 40% and 80% of the default DQN network. The network is trained on drop rates of 0.2, 0.4 and 0.8 over a set of 23 environments. Here, we run the agent for only 1 configuration per environment and compare with the Default DQN agent (5 configuration).

- **Hypothesis:** Drop out is used as a regularization technique in neural networks. Drop out function randomly deactivates the cells in the hidden layer which usually prevents the neural network from Overfitting. Here, we set the drop out rates of 0.2, 0.4 and 0.8 where we randomly deactivate 20%, 40% and 80% of the default DQN. Hence, we aim to decrease the number of activated neurons to 20%, 40% and 80% during training and check the influence of the DQN network under varying rates of drop out. Here, we want to validate, if the generalization of the agent takes a spike, by using varying rates of the drop out function.

- **Discussion:** In Figure 5.15 we have a radar plot showing us the default DQN agent along with varying rates of drop out to 0.2, 0.4 and 0.8. In the Figure 5.15, we can clearly see the default DQN with any drop out outperforms the drop out networks almost in every dimension expect the memory dimension. Strangely, the drop out 0.2 performs poorly on all the dimensions which gives a score of 0, but the other drop rates 0.4 and 0.8 although performing poorly in the all dimensions, perform better than default DQN in terms of memory. In order to dig a bit more deeper
Figure 5.15: Radar Plot: DQN with drop out rates of 0.2, 0.4 and 0.8

Figure 5.16: Histogram: DQN with drop out rates of 0.2, 0.4 and 0.8
we also generated a histogram of the profiles. In the Figure 5.16 we can see the difference in the bsuite scores of default DQN and drop out DQN networks. Both the drop out networks have a 1.0 bsuite score in memory length environment and hence the spike in memory dimension. As we have executed the DQN agent with drop out rates for only 1 configuration, more research needs to be done in this topic to understand the behaviour of the agent and the influence of drop out rates on them.

Noisy DQN:

- **Set-Up:** This experiment is inspired by the introduction of noisy layers within the underlying network of the DRL agent, in order to increase the exploration. Here, we introduce the noisy layers on top of the DQN network provided by the bsuite library\[FAP+18\]. Here, we run the agent for only 1 configuration per environment and compare it with the Default DQN agent\(5\) configurations.

- **Hypothesis:** The concept of Noisy Layer is inspired by the work of the researchers who worked on the noisy layers and introduced Gaussian Noise in the architecture to improve the exploration of the agent and create a more robust agent\[FAP+18\]. The concept of the Noisy DQN is discussed in the Subsection 3.2.1 under Noisy agent. Here, we hypothesize that the addition of the noisy layers to the default DQN would increase the exploration of the agent.

- **Discussion:** In Figure 5.17 we have a radar plot showing us the default DQN agent vs the Noisy DQN agent with noisy layers. In the Figure 5.17, we can that the Noisy DQN is not only at par with default DQN at certain dimensions but also has outperformed the default DQN at some dimensions. The noisy agent has
a better profile than the Default version in terms of Exploration and Memory. As hypothesized, the Noisy agent was expected to perform well at Exploration but it also performs well in the Memory capabilities as well. It has a poor score w.r.t to the default version only in the Scale dimension.

To analyse the differences in the bsuite scores of the agents we went deeper to one more granular level and generated the histogram representing the agent scores over the entire set of bsuite environments. We can see in the Figure 5.18 that the Noisy DQN agent has a bsuite score for environments like memory length and deep-sea which gives contributes to the increase in performances for memory and exploration dimensions in the profile of the agent. As we have executed the Noisy DQN agent with noisy layers rates for only 1 configuration, more research needs to be done on this topic to understand the behaviour of the agent and the influence of noisy layers on dimensions such as Exploration and Scale.

5.2.2 A2C Agent

Above we saw the experiments related to the architectural changes in the DQN agent. Here, we take a look at the experiments that are targeted towards the changes in the architectures of the A2C agent.

As already mentioned, due to the configuration issues we did not run the default A2C agent for the overall set of 20 configurations. Hence, we will refer to the A2C agent Figure 5.1 from all the bsuite agent profiles mentioned at the start of this chapter. In this section, we will only tune the network size of the A2C agent in an attempt to check its influence on other dimensions.

Network Sizes:

- **Set-Up:** In this experiment, we tune the network size of the agent by reducing to half its default value from 64 to 32 and also by doubling its value to 128 from 64. Here, we run the agent for only 1 configuration and compare it with the Default agent.
Figure 5.19: Radar Plot: A2C with network sizes of 32 and 128

- **Hypothesis:** The theory behind this experiment is mentioned in the Subsection 3.2.1 under the A2C agent. The different sizes of networks learn different intermediate features from the inputs. So we targeted the size of the network in this agent in an attempt to see the influence of the network size on the profile of the agent.

- **Discussion:** In Figure 5.19 we have a radar plot showing us the values of an A2C agent with network sizes 32 and 64. In Figure 5.13, we can only see one radar plot because the values of both the networks are the same. In this experiment, we hypothesized to see changes in the behaviour of the agent for different sizes of networks in the agent. We have an interesting phenomenon in this scenario where there is no change of value in the profiles of different sizes but both the radars have a different profile than the default setting. As compared to the default setting which can be seen in Figure 5.1 the A2C network with sizes 32 and 128 perform poorly w.r.t to the generalization and the exploration and it has only performed better than the default setting in the credit assignment. Again, as we execute the agent only one configuration more future works need to be in this aspect to validate the results and understand the behaviour of the agents.

5.2.3 Summary of RQ2 Exp’s:

Till now, we have seen all the experiments related to the second research question. For this set of experiments, we have majorly targeted the DRL algorithms implementations with bsuite agents such as DQN and A2C. We further divided the agents in terms of the experiments done on the agents. In each of the experiments, we saw the set-up of the experiment, hypothesis and finally a discussion section based on the results of the experiments. In this section, we are just summarizing what we have seen in this section in the form of the table Table 5.2.
5.3 Experiments: Research Question 3 (Hybrid Agents)

This section contains all the experiments which were aimed at research question 3. Here, the experiments are targeted to test the hybrid agents which we created using the different components of various RL algorithms. We have created 2 hybrid agents viz. A2C Hybrid and R2D2 Hybrid. A2C Hybrid is a combination of BootDqn \cite{OBPV16}, Critic driven actor loss \cite{Ste19} and A2C agent. R2D2 Hybrid is a combination of BootDqn and R2D2 agents. The details of these agents are given in brief in the Chapter 3, Subsection 3.2.1 under A2C and R2D2 agent.

Under this section, we have presented the profiles of both the hybrid agents. We have presented a list of experiments under each agent where we test the tuning of the different hyperparameters and their influences on the newly created hybrid agents.

### 5.3.1 A2C Hybrid Agent

Till now, we have seen the various types of RL algorithms and the experiments related to the hyperparameter values changes or the architecture structure changes of these algorithms. Now, in this section, we are using the Hybrid Agents which we have created by combining the different components and ideas from various RL algorithms. In this section, we will see all the experiments related to the A2C hybrid agents.

**Critic Driven Loss:**

- **Set-Up:** In this experiment we are going to see the profile of the A2C hybrid agent which is based on critic driven loss. Here, we use an actor ensemble of 20 networks along with the network priors. We executed our hybrid agent on all the 20 configurations of the entire 23 sets of environments to get a clear and concrete profile of the hybrid agent.

- **Hypothesis:** The theory behind this experiment is mentioned in the Subsection 3.2.1 under the A2C Hybrid agent. As already mentioned in the Chapter 3 Subsection 3.2.1 we have used the ensemble methods used in BootDqn and combined with the LSTM modules used in the A2C RNN to create this Hybrid agent. According to our hypothesize we have created this agent with a motivation to develop a powerful agent which performs well in Exploration and Memory dimensions.

- **Discussion:** In Figure 5.20 we have a radar plot showing us the values of an A2C Hybrid agent with 20 ensemble actors along with their network priors. In Figure 5.20, we can see that the Hybrid agent does not improve its performance.
on the exploration dimension and memory dimensions but the bsuite score is also poor in the dimensions like Basic and Credit assignment. We hypothesized that the hybrid agent would have better exploration and memory scores but we created a Hybrid agent which has a decent profile. We consider this as a basic foundation, where researchers can examine the various components of the various agents and create more powerful agents as per their requirements. As we ran the agent for 20 configurations, the agent profile is concrete. Due to the time and resource constraints we could not dig deeper in the analysis of the components and test all the possible reasons for the poor profile but none the less we share our findings in this report.

**Linear Layer Ensemble:**

- **Set-Up:** This experiment is a variation of the above A2C Hybrid agent. Here, the only difference in the setting is the use of just a single layer on top of the actor. Here, we too use the ensemble but with size 1, i.e. we just use a simple network of one layer on top of the actor to check its profile. Here, we execute the agent only on 1 configuration of the 23 bsuite environments.

- **Hypothesis:** The theory behind this experiment is mentioned in the Subsection 3.2.1 under the A2C Hybrid agent. The hypothesis of this experiment is the same as the previous experiment. Here, we have used just a single linear layer on top of the actor to test the hybrid agent on a simpler architecture.

- **Discussion:** In Figure 5.21 we have a radar plot showing us the values of an A2C Hybrid agent with a single linear layer. In Fig Figure 5.21, we can see that the Hybrid agent does not improve its performance on the exploration dimension but the bsuite score in the memory dimensions has improved significantly. As compared to the profiles of agents like A2C, BootDqn and even the A2C Hybrid this agent has a better performance in the memory dimension. Rest, it has a similar profile like Figure 5.20 for all the other dimensions. By looking at this observation, we
can hypothesize that instead of the ensemble, if a simpler network when used can increase the memory capacity of the agent. Again, as the agent has been executed for only 1 configuration, we can solidify the hypothesize and future work needs to be done to better understand the behaviour of the hybrid agent.

### 5.3.2 R2D2 Hybrid Agent

In this section, we are showcasing the second Hybrid agent that has been created by combining the different components and ideas from diverse RL algorithms like BootDqn and R2D2. Again, we attempt to capitalize the exploration and memory performances of the respective agents into a single most agent. Hence, we will see all the experiments related to the R2D2 hybrid agents in this section.

#### Ensemble Size 10:

- **Set-Up:** In this experiment we are going to see the profile of the R2D2 hybrid agent which is based on a combination of R2D2 implementation provided by acme and BootDqn implementation provided by bsuite. We have used an ensemble of size 10, mask as all ones instead of a binomial mask and prior scale of value 0.3. We executed our hybrid agent on all the 20 configurations of the entire 23 sets of environments to get a clear and concrete profile of the hybrid R2D2 agent.

- **Hypothesis:** The theory behind this experiment is mentioned in the Subsection 3.2.1 under the R2D2 Hybrid agent. As already mentioned in the Chapter 3 Subsection 3.2.1 we have used the ensemble methods used in BootDqn and combined with the LSTM modules used in the R2D2 agent to create this Hybrid R2D2 agent. The hypothesize of this hybrid agent is the same as above, to develop a powerful agent which performs well in Exploration and Memory dimensions.

![Figure 5.21: Radar Plot: Hybrid A2C (Linear Layer)](image)
Figure 5.22: Radar Plot: Hybrid R2D2

Figure 5.23: Histogram: Hybrid R2D2
Discussion: In Figure 5.22 we have a radar plot showing us the values of an R2D2 Hybrid agent with 10 ensemble networks along with their network priors. In Figure 5.22, we can see that the Hybrid agent does not improve its performance on the exploration dimension and the memory dimensions. It also performs poorly in all the dimensions.

To get a deeper insight of bsuite scores in the respective bsuite enthronements we have also generated the Figure 5.23. We can see the overall histogram on how the hybrid R2d2 agent performed all over the environments where it has performed poorly throughout. We believe that this is only one representation of the hybrid agent, with proper hyper tuning and a combination of different elements like binomial masks, prior scale values and epsilon decays we could have developed a more powerful agent with better results. Due to the time and resources, we could test the following scenarios.

We consider this as a basic foundation, where researchers can examine the various components of the various agents and create more powerful agents as per their requirements. As we ran the agent for 20 configurations, the agent profile is concrete. Again, due to the time and resource constraints we could not dig deeper in the analysis of the components and test all the possible reasons for the poor profile but none the less we share our findings in this report.

5.3.3 Summary of RQ3 Exp’s:

Till now, we have seen all the experiments related to the second research question. For this set of experiments, we have majorly targeted the DRL algorithms implementations with bsuite and acme agents such as A2C, BootDqn and R2D2 and created a Hybrid version of them. We further divided the agents in terms of the experiments done on the agents. In each of the experiments, we saw the set-up of the experiment, hypothesis and finally a discussion section based on the results of the experiments. In this section, we are just summarizing what we have seen in this section in the form of the table Table 5.3.

<table>
<thead>
<tr>
<th>Agents</th>
<th>Framework</th>
<th>Descriptions</th>
<th>Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2C</td>
<td>Hybrid</td>
<td>Critic Driven loss</td>
<td>Performs Poorly</td>
</tr>
<tr>
<td>A2C</td>
<td>Hybrid</td>
<td>Linear Layer Ensemble</td>
<td>Improves Memory</td>
</tr>
<tr>
<td>R2D2</td>
<td>Hybrid</td>
<td>Ensemble 10</td>
<td>Performs Poorly</td>
</tr>
</tbody>
</table>

Table 5.3: Summary of RQ3 Exp’s

Therefore, this brings an end to the longest chapter, where we break down the experiments in terms of research questions defined in Chapter 3. The entire list of the experiments is designed in a way to circumvent and address the given research questions. We divided this chapter in terms of the experiments related to their respective research questions i.e. RQ1 to RQ3. Furthermore, we have grouped the experiments related to research questions under the agent to which it belongs. Also, we have followed a structure to define each experiment, where we discuss the set-up, the hypothesis and the discussion of the experiment.
which covers the configuration, the motivation behind the experiment and findings of the experiment. At the end of each research question-related experiment, we have also added a summary section, to summarize the relevant findings.

Hence, we go through the different DRL agents and their respective hyperparameters, their tuning and our findings at the end of which. Then, we switch to the architectural changes of the DRL agents and our findings w.r.t to the experiment list. Finally, we shared the results of our own created hybrid agents and discussed more the profiles of the agent. Although the results aren’t great we have a positive impression of the hybrid agents as it becomes a stepping stone to dive into this field of DRL agents which has been unprecedented and we are proud to present our findings in the given field.
6 Conclusion and Future Work

In the above chapters, we saw the introduction of the DRL agents and why there is a need to benchmark the modern-day RL algorithms. Then, we switched to the necessary prerequisites of RL, DRL, and their respective algorithms, we also went through a literature overview of the related work in the field of RL benchmarking. Later, we discussed the Design of our project and the Experimental setup providing details about the Bsuite Environments, the software used, and their configurations. In the later sections, we saw the Evaluation section where the results and findings of our experiments were discussed in brief. Finally, this brings us to the last chapter of this report, i.e. Conclusion and Future Work. In this chapter we will summarize our work and our findings, we will discuss the threats left unchecked due to time and resource limitations, and talk about the potential future work that can be done on top of our work to benefit the RL community.

6.1 Conclusion

In this report, we have presented the profiles of the different DRL agents presented by the Bsuite and Acme frameworks. We made changes to the different hyperparameters of the DRL agents provided by the Bsuite and Acme frameworks and we observed their influences on the profiles of the agent. During this, we observed improvements in the profiles of the agent which encouraged us to take it one step further and make changes in the architectures of the simple DRL algorithmic implementation provided by the respective frameworks. We made changes in the network sizes, architectures of the DRL and got positive improvements in the dimensions of exploration and memory. Finally, we took another stride in our research and created a couple of hybrid agents, in order to test if we can successfully create a DRL agent combining the different strategies used in various RL algorithms. We could successfully develop 2 hybrid agents viz. A2C Hybrid and R2D2 Hybrid agents. We tested the agents on the Bsuite environment and created a profile of these hybrid agents too. Although the profiles are not that encouraging, we feel this could lay a foundation in the RL community and the researchers could use this intuition of the profiles of the agents to create modern-day more powerful agents that perform well in all the dimensions and have a complete profile.

6.2 Threats to Validity

There are several hypotheses that need to be validated in this work. As many of the results shared in the evaluation section are based on only one configuration execution. Running the different experiments for all 20 configurations (i.e., random seeds) could
take a lot of time, especially when using components like ensembles with random network priors making the architectures and computations complex. One such agent has to then run 460 configurations in total considering 20 configurations per environment for 23 environments. Due to such a time-consuming process, we could not run all the experiments for all the 20 configurations, used in the original work. Therefore, leaving a lot of room for validation of our results and our hypothesis which are executed for 1 configuration mostly, instead of 20 configurations. In this work, we haven’t executed all the experiments for 20 configurations as we included a variety of experiments because we aimed at providing a larger width of experiments encompassing our research questions. Researchers can then select the specific experiments from our research as per their domain requirements and dig deeper into the following. Hence, the experiments executed for 1 configuration need more execution effort to reproduce the results and get an intuition about the stochastic changes in behavior of the agent stemming from different random initial parameters.

6.3 Future Work

Finally, let’s discuss the future work and the scope of the future work related to our project. Let’s first begin with the DRL frameworks used in the project. Bsuite was released in recent years around 2019 and there is continuous ongoing research happening in the tool, where the authors have prepared a committee and planned to meet annually in the NeurIPS conference to add new environments to the list of 23 environments. In case there is an addition of new improved environments, one has to take into consideration all those updates when working on this project. Rest, in our work, we need to do future work in terms of validating the results of our experiments where the agent has been executed on only 1 configuration. As of now, we suspect the agent’s behavior to be the same as mentioned in the evaluation section but one needs to validate the results and run the agent for 20 configurations. Apart from validating the results, future work needs to be done w.r.t to the hybrid agents created in this work. The behavior of the hybrid agents is poor for the overall set of environments, more future work needs to be done to find out the possible hyperparameter combinations and architecture tuning needs to be done to achieve the ultimate goal of creating powerful agents. Apart from the model-free agents which was the scope of our work, researchers can extend this concept of profiling of agents and creation of modern-day powerful RL algorithms for the Model-Based algorithms as well.
### Bibliography


