# IMPROVING MODEL-FREE ALGORITHMS IN DEEP REINFORCEMENT LEARNING

EMTIAZ SAMAD

MASTER OF SCIENCE IN DATA SCIENCE
THE UNIVERSITY OF BATH
2021-2022

# IMPROVING MODEL-FREE ALGORITHMS IN DEEP REINFORCEMENT LEARNING

Submitted by: Emtiaz Samad

## COPYRIGHT

## DECLARATION

## ABSTRACT

Over the recent years, many reinforcement learning agents have been developed to solve Atari2600 games with outstanding performance. This paper investigates how one of the most successful distributed RL agents called R2D2 can be improved by combining design choices from other existing agents. The paper also delves into the background of R2D2's predecessors and explains each component of the agent from the ground up to provide a thorough understanding. Notable improvements are made to the existing agent by combining the model architecture of R2D2 with techniques inspired by an existing agent called D2RL. The paper illustrates how adding more layers paired with dense (skip) connections to the model architecture can result in increased sample efficiency and enable more stable learning.

# CONTENTS

# LIST OF FIGURES AND TABLES

## ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my research supervisor, Mr. Ken Cameron, for providing his expertise and supporting me in undertaking this project. Were it not for his guidance, the outcome of this project would not be possible. I would also like to acknowledge the University of Bath's Department of Computer Science for providing access to the Hex GPU cloud, for which I am indebted to for enabling me to conduct this research. Finally, I would like to give credit to Michael Hu, the developer of Deep RL Zoo, for providing the software which serviced as a platform for my research.

# 1 INTRODUCTION

## 1.1 PROBLEM DESCRIPTION

Over the years, there has been an increasing emergence in applications of Deep Reinforcement Learning (DRL), most notably in autonomous vehicles and projects such as DeepMind's *AlphaZero*, a program with the aim to solve the games of *Chess*, *Shogi* and *Go* (Moreno-Vera, 2019). Reinforcement Learning is a concept in which an agent takes actions in an environment to maximise its cumulative reward. Agents are incentivised by rewards for good actions or punished otherwise (Sharma, Prateek and Shina, 2013). Deep Reinforcement Learning combines the idea of Reinforcement Learning with neural networks to allow reinforcement learning methods to be applied to larger problems. DRL has recently found major success in Robotics and solving *Atari* games with variations of *DQNs* (Deep Q-Networks; Mnih et al., 2015) which is primarily built upon Q-Learning, an algorithm which stores Q-values of state-action pairs containing the sums of all possible rewards. DQN solves the infeasibility of storing state-action pairs in large environments by creating a Neural Network that predicts the Q-value based on the input. This is much more tractable than storing every possible state-action pair (Gitau, 2018). These recent breakthroughs in DRL and handling large state spaces serve as a motivation to discover powerful algorithms which solve all complicated general reinforcement learning problems.

So how is solving games with Reinforcement Learning useful? Games provide excellent domains for AI research since they allow agent-environment interactions to be modelled through interesting and complex problems (Shao et al., 2019). More importantly, games can serve as a benchmark to test the intelligence of RL agents in a safe, controlled environment in the hopes of translating successful implementations of RL to purposeful applications such as training robots or optimising data centres. Recently, there has been growing popularity in testing DRL algorithms in games from the *Arcade Learning Environment* (ALE), consisting of an emulator and a large selection of retro *Atari2600* games (Guo et al., 2014). Despite the relative simplicity of the *Atar*i games by modern standards, they provide huge challenges in diversity and perception. Diversity in this case refers to the differences in how the games are played, the goal of the game, and most recently game difficulty (Machado et al., 2018). This diversity is instrumental in its popularity as it tackles the issues of model overfitting. Essentially, the ALE serves as a benchmark set of tasks to measure Artificial General Intelligence and is appropriate for three main reasons: '(i) varied enough to claim generality, (ii) each interesting enough to be representative of settings that might be faced in practice, and (iii) each created by an independent party to be free of experimenter's bias', (Badia et al., 2020, p.1). Note that the 'goal isn't to develop systems that excel at games, but to use games as a stepping-stone in developing systems' that reach and surpass human-level performance in a wide range of challenges (Badia et al., 2020).

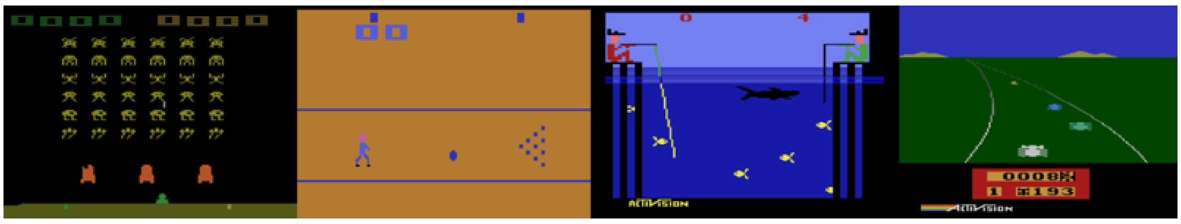


Figure 1.1: *Atari 2600* Games from left to right: *Space Invaders*, *Bowling*, *Fishing Derby* and *Enduro* (Machado, 2016).

Substantial progress has been made since DQN's success in solving various Atari Games through the creation of both model-based and model-free algorithms such as MuZero (Schrittwieser et al., 2020) and *Agent57* (Badia et al., 2020) respectively. These algorithms seek to address the underlying issues which DQN posed such as interminable learning times and suboptimal mean performance across all games in the *Atari* collection (Hessel et al., 2017). To briefly summarise the strengths of the algorithms stated above, MuZero astronomically surpasses average human performance on some games yet performs disastrously in others, whereas Agent57 has proven to surpass human performance in 57 *Atari* games but is still unable to achieve optimal performance in all of the games. In fact, all current algorithms are far from achieving optimal performance in some games (Toromanoff, Wirbel and Moutarde, 2019). The strengths of MuZero do not align with the aim of my project as it is a model-based algorithm which performs excellently on a selection of games, as opposed to all. Therefore, I will not discuss this algorithm any further unless I am comparing its performance. Additionally, the algorithms stated above are extremely complex and have many components which make them immensely hard to replicate. Therefore, I will be focusing on improving relatively simpler model-free algorithms such as *R2D2* (Recurrent Experience Replay in Distributed Reinforcement Learning; Kapturowski et al., 2019). This leads to the focus of my research, how can model-free algorithms such as R2D2 be improved?

## 1.2 RESEARCH OBJECTIVES

### 1.2.1 PRIMARY AIM

My overarching objective is 'to implement a model-free Deep Reinforcement Learning algorithm which produces remarkable performance in Atari games without alterations in parameters or the underlying architecture and improves upon its related predecessors'.

Having carried out extensive research on the leading solutions to the identified problem, I will briefly evaluate the utility and shortcomings of each solution and attempt to replicate some of the models. From the research I have conducted, I will attempt to improve upon existing models by analysing areas in which previous authors have stated that their implementations could be improved. Understandably, the tasks may be very complicated so I will implement my RL agents in an incremental fashion to allow for multiple exit points. In consideration of the complexity of the task at hand and the time constraint of the project, I have established the following milestones which I hope to achieve. I will focus on progressively implementing DQN and R2D2. Upon completing these tasks, I will attempt to improve upon the shortcomings of R2D2 by exploring areas such as enhancing the representations that SOTA model-free algorithms use for exploration (Badia et al., 2020).

### 1.2.2 DELIVERABLES

By conducting this project, I aim to contribute to further research on the performance of existing model free algorithms, improve upon the leading solutions in my area of focus and provide additional insight into how these solutions may be further improved. This contribution will made through assessing performances of viable solutions (by replicating previous work) and hopefully reporting improvements on existing research. Comparisons in performance will mainly be assessed by producing plots of mean performance of the best solutions over a selection of games. These results will be analysed so that strengths and weaknesses of the algorithms can be identified.

## 1.3 BRIEF OVERVIEW OF METHODOLOGY

Throughout my research, I will be emulating the *Atari2600* console to play a selection of games with RL agents. I will access this emulator through the *Arcade Learning Environment* framework which currently supports over 50 games (Bellemare et al., 2013). The Arcade Learning Environment can be found within *OpenAI* Gym, a toolkit for the development and comparison of RL algorithms. *OpenAI* Gym contains a collection of benchmark problems ranging from classic control problems such as balancing a pole on a cart to more complex problems like those found in solving *Atari* games (Stapelberg and Malan, 2020).

To grasp an understanding of the toolkit, I will first implement simple RL algorithms on a selection of classic control and toy text problems such as *CartPole* and *FrozenLake* respectively. This will allow me to understand how the environments behave, the different methods available and how to interpret the observations. Practicing with the toolkit will also provide me with knowledge on how to debug errors, which is useful to know in early stages since it will become progressively harder to determine bugs as the games become more complicated. Once I've familiarised myself with the toolkit, I will then begin to recreate the fundamental DRL models as seen in previous research in attempts to reproduce their success. Upon reaching a solid comprehension of the toolkit and having successfully implemented high performing DRL algorithms on simple *Atari* games, I intend to discuss how I could potentially improve my models with my supervisor. Once I've established proof of concept on the simpler games, I will attempt to implement my model on some of the more complex games in the *Atari* suite. I recognise that accomplishing this task will be a difficult feat and I could be hindered by time constraints, so I have considered the steps mentioned above as possible 'exit points' in my research as results to demonstrate if I am unable to achieve my objective.

# 2 LITERATURE AND TECHNOLOGY SURVEY

## 2.1 INTRODUCTION

The success found by implementing DQNs to obtain superhuman levels of performance in *Atari2600* has resulted in heavy research in improved variants. These variants seek to resolve the shortcomings of traditional Deep Q-Learning such as the requirement for millions of training steps in order to achieve human levels of performance (He et al., 2017). Additionally, the performance of DQN falls short in comparison to planning agents but outperforms the latter on time, so there exists an opportunity to fill this performance gap (Guo et al., 2014).

It became apparent that the original DQN method sometimes performed poorly due to overestimating action values, so to improve the stability, Double Q-Learning (Hasselt et al., 2015) was combined with neural networks (Double-DQN) with the goal to alleviate the overestimation. In principle, *DDQN* uses separate Q-Networks for action selection and Q-function value calculation. The target's greedy action is chosen initially, followed by the computation of the target value on a set of parameters from a prior iteration (He et al., 2017). Other notable advances include the previous state-of-the-art *prioritised experience replay* (Schaul et al., 2016) which initially assumes all transitions present in the replay memory may have varying importance. The main idea is to increase replay probability of experience tuples based on the magnitude of their temporal difference error (Hosu and Rebedea, 2016). Furthermore, the solutions mentioned in the introduction such as R2D2 and its successor Agent57 are all built on improvements made to the original DQN model. All these algorithms will be discussed more in depth in the following subsections.

The aforementioned variants are all considered model-free RL algorithms and were predominantly the conclusive approach until recently. There has been growing interest in the implementation of model-based RL methods for Atari since it excels in requiring substantially fewer interactions to solve a game. A primary example of this concept is demonstrated in Simulated Policy Learning (*SimPLe*), a model-based DRL algorithm based on video prediction models which rely on limited data of 100k agent-environment interactions to outperform current state-of-the-art model-free algorithms (Kaiser et al., 2020). Despite the success of model-based approaches in recent years, my main focus is improving the performance and efficiency of model-free DRL algorithms in general gameplay, so exploring model-based algorithms is not currently an avenue I wish to take. In this project, I will strictly limit my research focus to model-free RL algorithms and briefly mention model-based concepts to compare results.

## 2.2 CONCEPTS IN REINFORCEMENT LEARNING

### 2.2.1 AGENT-ENVIRONMENT STRUCTURE

Principally, RL is based on an agent performing actions in an environment in order to maximise a scalar reward as illustrated by Figure 2.1.

There are typically five key components of the reinforcement learning problem:

1. Environment: The world in which the agent operates.
2. State: The current state of the agent in environment.
3. Reward: Feedback received from the agent taking an action in the environment.
4. Policy $\pi(s, a)$: Comprises the suggested actions that the agent should take for every state.
5. Value $q(s, a)$: Reward that agent would receive by taking an action in a specific state.

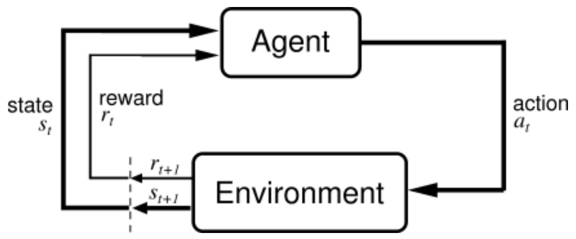


Figure 2.1: The *agent-environment* interaction in Reinforcement Learning (Sutton and Barto, 2014).

### 2.2.2 MODEL-BASED AND MODEL-FREE LEARNING

RL can be subdivided into two key categories: model-based and model-free learning. In model-based learning, a model of the environment is constructed which stores state transitions and expected rewards. These components are used to predict the next state and the reward gained from the transition respectively. Such models are represented by a Markov Decision process (MDP). Once a model is constructed, a planning algorithm is applied to calculate the optimal value or policy of the MDP. Alternatively, model-free algorithms do not use the state transition probability distribution which are associated with MDPs. A prime example of a model free algorithm is Q-Learning (Watkins and Dayan, 1992).

### 2.2.3 Q-LEARNING

Q-Learning is a model free algorithm which learns the action-value of a particular state. In any finite MDP, Q-Learning will always converge and find the optimal policy if given infinite exploration time. Fundamentally, Q-Learning incrementally stores state-action values in a Q-Table which is updated at every time step $t$, such that the agent selects the action $a_t$ producing a reward $r_t$ and transitions to a new state $s_{t+1}$.

The Q-Table is updated using an adaptation of the Bellman Equation which is described by the following formula:

$$Q^{new}(s_t,\ a_t) \leftarrow\ Q(s_t,\ a_t) + a \cdot (r_t + \gamma\ \cdot \max_a Q(S_{t+1}, a) -\ Q(s_t,\ a_t)$$

where $a$ is the learning rate, and $r_t$ is the reward for transitioning from the current state $s_t$ to the next state $s_{t+1}$.

### 2.2.4 DEEP RL AND DEEP Q-NETWORKS

Environments with large state or action spaces are an impediment to Q-Learning as they make it intractable to learn Q-values for each state and action independently. Deep Reinforcement Learning solves the issue of high dimensional MDPs by combining Deep Learning with Reinforcement Learning techniques. This if often done by representing a learned function such as the policy $\pi(s, a)$ as a neural network and developing appropriate algorithms that perform well in conjunction with this policy. This was successfully implemented in DQN (Mnih et al., 2015), which used convolution neural networks to estimate state-action values. At each step $t$, an action is selected through an $\epsilon$-greedy policy and a transition $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ is stored in a replay memory buffer which contains the last million transitions.

The parameters of the neural network are suitably tuned using gradient descent to minimise the loss:

$$(R_{t+1} +\ \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') -\ q_\theta(S_t, A_t))^2$$

of which the gradient is backpropagated into the online network with parameters $\theta$. A copy of the network called the target network with parameters $\overline{\theta}$, evaluates the policies selected by the online network. The conjunction of Experience Replay and target networks enables relatively stable learning and resulted in optimal performance on multiple *Atari* games.

## 2.3 IMPROVEMENTS ON DQN

### 2.3.1 PRIORITISED REPLAY AND DOUBLE DQN

Prioritised Experience Replay samples transitions from which there is more to learn in increasing frequency, unlike standard Experience Replay which samples uniformly from the replay buffer. Transitions are sampled with the probability $p_t$ corresponding to the last absolute temporal-difference error:

$$p_t \propto\ \left|R_{t+1} +\ \gamma_{t+1}\ \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') -\ q_\theta(S_t, A_t)\right|^\omega$$

such that $\omega$ determines the distributional shape. Recent transitions have more bias as new transitions are added to the replay buffer with maximum priority.

Double DQN seeks to solve the issue of overestimating values which is typically present in all Q-Learning methods by separating the selection and evaluation of actions so that a different Q-Network can be used for each step. This is achieved by implementing a target Q-Network which evaluates the actions selected by the online Q-Network. The framework of Double DQN is built on the Double Q-Learning algorithm (Hasselt et al., 2015) which produces two sets of weights, $\theta$ and $\theta_t$, from two value functions which are learned by randomly by assigning experiences to randomly update either of the value functions. One set of weights is used to determine the greedy policy while the other determines its value for each episode. However, Double DQN replaces the weights of the second network $\theta_t^-$ with the weights of the target network $\theta_t'$. Evidently, Double DQN retains the benefits of Double Q-learning, while maintaining the structure of the DQN algorithm. Essentially, the Double DQN algorithm is identical to DQN with a minor, albeit important modification; Double DQN replaces the target of DQN with

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname*{argmax}_{\alpha} Q(S_{t+1}, \alpha; \theta_t), \theta_t^-\right).$$

### 2.3.2 MULTI-STEP LEARNING AND DUELING NETWORKS

The n-step return from a given state $S_t$ can be defined as

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

such that the multi-step DQN variant is defined by the loss:

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2.$$

Generally, faster learning is achieved if multi-step targets with a suitably tuned $n$ are used (Sutton and Barto, 1998).

For many states, it is unnecessary to estimate the value of each action. This is because in some states, it is highly important to know which action to take but in other states, some actions may lead to insignificant repercussions. *Dueling Networks* address this by combining two streams of value and advantage functions, to produce a single output Q-function, such that its output is a set of Q-values for each action (Wang et al., 2016).

### 2.3.3 DISTRIBUTED LEARNING

Distributed learning changes the landscape of reinforcement learning as it introduces a new component to the traditional structure. Distributed RL typically consists of the environment, the learner and the actor which communicate with each other. This enables many distributed RL methods to exploit parallelism in the actor or learner, thus allowing agents to be run on several machines simultaneously. This practice allowed researchers to develop distributed architectures which produced agents that ‘can learn from experience faster, leverage exploitation strategies, and become capable of learning a diverse set of tasks simultaneously’ (Samsami and Alimadad, 2020, p.1).

## 2.4 RECURRENT REPLAY DISTRIBUTED DQN

The Recurrent Replay Distributed DQN (R2D2) algorithm incorporates many of the concepts mentioned above and builds upon previous distributed architectures such as *Ape-X* (Horgan et al., 2018). Much like Ape-X, R2D2 utilises prioritised experience replay amongst other extensions on DQN such as dueling networks and n-step Double Q-Learning. However, R2D2 incorporates recurrent architecture into Ape-X immediately after the convolutional stack. The algorithm doesn’t store the regular transition tuples $(s, a, r, s')$, but instead stores sequences of $(s, a, r)$ in replay with a fixed length of 80. Adjacent sequences overlap by 40 steps and never cross episode boundaries. In R2D2, value estimates and targets are generated from both the online and target networks on the same sequence. R2D2 maintains a similar distributed training architecture as Ape-X, which separates learning from acting by allowing multiple ‘actors’ running in parallel instances to feed experiences into a distributed replay buffer, which a ‘learner’ would then sample training batches from. The learner would use these samples to update the network and priorities of the experiences. Consequently, the actors’ networks would be updated following the latest network parameters from the learner.

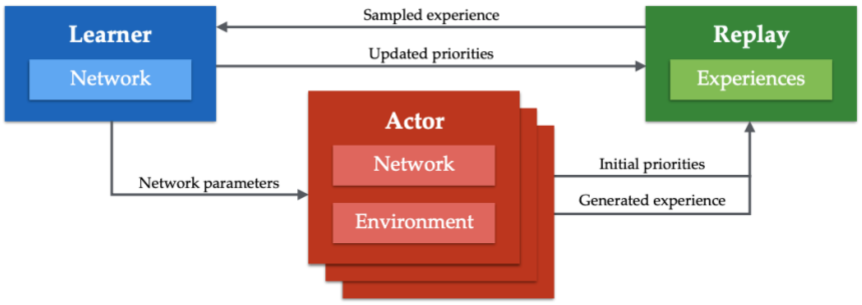


Figure 2.2: A diagram of a distributed learning architecture, in this case Ape-X (Horgan et al., 2018).

### 2.4.1 LONG SHORT-TERM MEMORY (LSTM)

LSTMs are a type of Recurrent Neural Network (RNN) which were originally designed to address the flaws of RNNs when learning time-series with long-term dependencies (Bakker, 2001). Essentially, LSTMs seek to overcome vanishing gradients and exploding gradients. With regards to RL, they can provide better judgement in POMDP (Partially Observable Markov Decision Process) systems. The structure of an LSTM contains input $x_t$, previous output $h_{t-1}$, and previous cell memory $c_{t-1}$, of which the latter two combined make the LSTM state. It would pose an issue if transitions were stored regularly as tuples of $(s, a, r, s')$, because R2D2 stores consecutive transitions as segments in the replay buffer. This means that the LSTM would have no initial state. R2D2 remedies the weakness of a zero-start state by introducing the *Stored State* strategy (Kapturowski et al., 2019). This is accomplished by storing the LSTM state in the replay buffer and using it to initialise the network during training. This technique alone fails to solve the initial start state problem because the stored LSTM state is computed by the previous network, hence the value produced is different to computing the state with the current network.

R2D2 addresses stored LSTM staleness by using *Burn-in* (Kapturowski et al., 2019), which hypothetically allows the network to be in a better initial state before it is required to produce accurate outputs. This strategy mitigates the effects of zero-start and stored-stale problems by producing a start state with a portion of replay sequence used for unrolling the network and updating the network with the remaining portion.

### 2.4.2 VALUE FUNCTION RESCALING

R2D2 replaces reward-clipping with invertible value function rescaling to reduce the variance of the target, in the form:

$$h(x) = sgn(x)\left(\sqrt{|x|+1}-1\right) + \epsilon x,$$

$$h^{-1}(x) = sgn(x)\left(\left(\frac{\sqrt{1+4\epsilon(|x|+1+\epsilon)}-1}{2\epsilon}\right)^2 - 1\right),$$

giving the following n-step targets for the Q-value function:

$$\hat{y}_t = h\left(\sum_{k=0}^{n-1} r_{t+k}\gamma^k + \gamma^n h^{-1}\left(Q(S_{t+n}, a^*; \theta^-)\right)\right), \quad a^* = \operatorname*{argmax}_a Q(S_{t+n}, a; \theta)$$

where $\theta^-$ is copied from $\theta$ every 2500 learner steps.

This reward function provides better reward scaling as it allows the agent to differentiate the rewards gained from performing different actions. For example, in the game *Bowling*, the agent does not differentiate between striking a single pin or all pins because both actions would result in a positive clipped reward of 1.
R2D2 is one of the algorithms at the forefront of general game-playing as it uses a single neural network architecture and constant hyper parameters across all games, achieving super-human performance in 52 out of 57 games. At the time of its formation, R2D2 achieved the highest reported results in a large selection of games, though it failed in games such as *Montezuma's Revenge* and *Pitfall*, which are considered to be hard exploration problems. R2D2 takes advantage of the key improvements built on DQN such as Prioritised Experience Replay, Dueling Networks, Double DQN, Distributed RL and pairs it with LSTMs.

## 2.5 R2D2's SUCCESSORS

### 2.5.1 NEVER GIVE UP (NGU)

*NGU* is the most notable algorithm in recent years to have shown significant promise in improving performance in hard-exploration games. The algorithm accomplishes this by augmenting the reward signal with internally generated rewards, which are sensitive to short-term novelty within episodes and long-term novelty across episodes. A family of policies for exploration and exploitation are learned with the goal to retrieve the highest score under the exploitative policy. The intrinsic rewards computed by NGU are defined as a combination of episodic and life-long novelty. The episodic novelty $r_t^{episodic}$ quickly disappears over the duration of the episode and is calculated by comparing observations to the contents of an episodic memory. The life-long $a_t$ novelty is computed by using a parametric model and it disappears gradually. Thus, the intrinsic reward can be defined as:

$$r_t^i = r_t^{episodic} \cdot min\{max\{a_t, 1\}, L\}, \qquad s.t.\ L: maximum\ reward\ scaling.$$

This reward encourages the agent to explore episodically, whilst leveraging long term novelty. At timestep $t$, NGU forms $N$ potential total rewards by adding $N$ different scales to extrinsic rewards of the environment, with the aim of learning $N$ corresponding optimal state-value functions associated with each reward function.

Despite the proven success in hard-exploration games, NGU fails at being the best general agent since it performs poorly in other games (similar to a random policy). A major issue of NGU is that regardless of its policy's contributions to learning progress, it collects a constant amount of experience. This allows NGU to be adapted to a more general agent, given the ability to adapt its exploration strategy. Another flaw of NGU is its instability and failure in learning an approximation of the optimal state-value functions, including simple environments.

### 2.5.2 THE CURRENT STATE-OF-THE-ART

As previously mentioned, the leading model-free high performing algorithm in general Atari game-playing is Agent57. This model serves to fill the gap between the peak performance provided by *MuZero* and the generality offered by R2D2. Agent57 builds on the NGU model, by proposing several important improvements such as 'i) using a different parameterisation of the state-action value function, ii) using a meta-controller to dynamically adapt the model preface and discount, and iii) the use of a longer backprop-through time window to learn from using the *Retrace* algorithm' (Badia et al., 2020, p.1).

### 2.5.3 CONTRIBUTIONS

The following information provides more depth to the improvements made on NGU to create Agent57:

1. Significant increase in training stability over a large range of intrinsic reward scales, due to a parameterisation of state-action value functions which decompose the contributions of intrinsic and extrinsic rewards.
2. Improved control of exploration/exploitation trade-off, by implementing an adaptive mechanism to select policies to prioritise during the training the process, called a meta-controller.
3. Solving the long-term credit assignment problem by improving stability, adjusting the discount factor dynamically and doubling the backprop-through time window, leading to exceptional long-term credit assignment and retaining performance in dissimilar games.

### 2.5.4 A COMMENTARY ON MODEL-FREE ALGORITHMS

It is evident that the state-of-the-art model-free DRL algorithms consist of many intricate components which have gradually formed a convoluted solution. As a result, it has become difficult to continue to enhance the current best performing algorithm, partially because it requires an advanced understanding of all the previous contributions made towards it. In fact, this concept coincides with *The Bitter Lesson* (Sutton, 2019) which essentially describes how over-engineered complex systems that are resource intensive can fail in the long-term for an approach in different direction. This calls for the need to seek a simpler and more elegant solution. This leads me to investigate a fundamental predecessor of the SOTA mentioned above called R2D2.

# 3 METHODOLOGY AND DESIGN

## 3.1 REPLICATING DQN: THE FIRST MILESTONE

In order to grasp all the technologies involved in this project, it was imperative that a relatively simple model was trained on several games to ensure that I had the necessary skills to utilise the provided environments effectively. The implementation of this agent had required knowledge of using the University of Bath's *Hex Cluster* and required the use of skills such as navigating the Linux OS through a CLI, SSH and port-forwarding. Furthermore, it allowed me to experiment with libraries such as *TensorBoard* which proved to be extremely useful as it enabled me to monitor training processes. The handling of experience tuples in the replay buffer required compression and decompression techniques. This involved compressing the experience tuples into bytes before pushing them into the buffer and decompressing such experiences when sampling batches. This reduced the memory consumption significantly and allowed me to use a much larger experience buffer. This solved an issue during training in which the program would be terminated for consuming too much memory.

A DQN agent was trained on the following games: *Pong*, *Breakout*, *BeamRider*, *Qbert* and *Montezuma's Revenge*. The selection of games were curated with the purpose of highlighting the performance of different agents in a variety of games. In particular, these games were chosen such that results could be assessed on both 'easy' and 'hard' exploration games, which are further categorised into *Human-Optimal*, *Score Exploit*, *Dense Reward*, and *Sparse Reward* games (Ostrovski et al., 2017). The architecture of the network model and agent remain consistent with DeepMind's DQN nature paper (Mnih et al., 2015), albeit some minor changes in the optimiser used (Adam in place of RMSProp). The frames were pre-processed by converting from RGB to greyscale, then cropped and resized to an 84×84 image. The last four frames were stacked upon being processed to produce the Q-function input which produce the input the neural network as an 84×84×4 image. This is followed by three hidden layers which convolve 32 filters of 8×8 with stride 4, 64 filters of 4×4 with stride 2 and 64 filters of 3×3 with stride 1 respectively, which are then flattened. This is followed by dense (fully connected) hidden layer which consists of 512 units. The output layer is also a dense linear layer which returns a single output for every valid action.

As mentioned previously, a new network with identical architecture and hyperparameters for the learning agent was used to train each game. The games were unmodified with the with the exception changes in the reward structure and life loss; a reward clipping function was utilised to clip the rewards to 1 (if positive), 0 (if 0) and -1 (if negative). This change in reward structure accommodates for the varying scale of scores across each game in exchange for calculating rewards of different magnitudes. The other change made was the handling of the loss of a life. Some games allow the player multiple lives. This information was used to terminate the episode when a life was lost. In some cases, these changes can improve the performance of the agent in particular games and also speed up the training process, however the agent can possibly perform worse as a result (Mnih et al., 2015). Other notable remarks include changes made to the replay buffer to handle a large number of experience tuples. More specifically, to handle the memory consumption of a replay buffer with a capacity of 1 million frames, experience tuples were compressed into bytes and then decompressed when a batch of size 32 was sampled. It should be noted that very large or small replay buffers could significantly hinder the performance of an agent. If the buffer is too small, it serves little purpose. Contrastingly, if the buffer is too large, the batched samples are likely to be uncorrelated which would increase the time required for the agent to learn (Zhang and Sutton, 2018). These are few of many nuances for choosing appropriate sizes for replay buffers which require investigation (Fedus et al., 2020). Further information detailing DQNs can be found in section 2.2.4. A comprehensive list of all the hyperparameters used can be located in the appendix (see Table A.1).

In consideration of the time given to complete this project, the agent was trained for approximately 2.5M steps on each game. This equates to 10M frames on the deterministic environments found within ALE, i.e., a fixed 4-frame skip per step. For each game, the training and evaluation procedures were designed to perform simultaneously such that every 20 episodes, the trained agent would play a single game with an $\varepsilon$-greedy policy where $\varepsilon = 0.05$. These test scores were then recorded to a summary, along with the train scores and losses of each run. A checkpoint containing the current saved weights of the network was created every 500,000 steps and at the final episode of training, of which an agent was tested to record a video.

## 3.2 TRAINING R2D2: THE SECOND MILESTONE

Unfortunately, due to time constraints and hardware limitations, I was unable to implement and train my own version of R2D2 on the chosen Atari games. This is primarily because R2D2 is requires a GPU and is also highly CPU intensive since it utilises a large quantity of actors to operate. This made it difficult to assess whether the code executed, and the agent performed as intended on the cloud environment  because of the lack of sufficient number of cores available for parallel processing (R2D2 typically 256-CPU actors and 1-GPU learner). Determining a working agent would require reducing the number of actors, which in turn greatly hinders performance (Horgan et al., 2018). Due to the reasons listed above, I prioritised running R2D2 on Pong because it can be solved in a short amount of time and produce good results. It is important for me to note that the code used was created by another individual and all credit goes to the author of this code. The agent used for my experiments belongs in a collection of DRL agents called *Deep RL* Zoo (Hu, 2022) and is licensed under the Apache 2.0 License. Also, the Atari games used in this implementation are of the 'NoFrameskip' variant.

During the experimentation process, I had tried various combinations of batch sizes and number of actors to obtain the best performance. This is because the agent was severely limited by the amount of VRAM available in the GPU (8GB). As a result, the GPU could only handle 6 actors with a batch size of 4. I had also reduced the size of the replay buffer however this did not appear to have a noticeable effect. Additionally, I had attempted to expedite the training to multiple GPUs with data parallelism to no avail. I aimed to solve this issue by forfeiting speed for functionality by allocating layers of the model to different GPUs. In theory, this performs slower than the previous solution as at any point, only a single GPU is occupied while the others are idle (Li, 2022). This method also requires outputs to be copied to a different device every time a new GPU is called between each layer, causing further decline in performance. This also proved to be unsuccessful. Despite my attempts to allocate enough memory to increase the batch size, R2D2 appeared to perform well on the reduced parameters. Distributed learning was performed using actors (environments) running in parallel, accompanied by one learner. The hyperparameters chosen were designed to fit the hardware, thus differ greatly from those used by DeepMind. Most notably, the replay buffer capacity, actor update frequency and target network update frequency were heavily reduced. The exploration value for the evaluation run was also reduced to 0.001 as opposed to 0.05, as used in DQN. R2D2 handles life loss differently to DQN as losing a life does not result in the termination of an episode. Unlike DQN, the training was performed separately to the evaluation run to reduce the time required to train the agent. To portray the performance of the agent, each actor was trained over 5M environment steps with the exception of Pong, which was trained for 2.5M steps. All agents were evaluated on 1M environment steps. A comprehensive list of all the hyperparameters used can be located in the appendix (see Table A.2).

The functionality of the software proved to be very useful in recording the statistics of the learner and actors. It allowed the monitoring of metrics such as network and target network updates, as well as episode returns across environment steps, all accessible through TensorBoard. R2D2 demonstrated that it is a very powerful algorithm, solving games such as pong in remarkably less time than agents such as DQN. Comparing my own implementation of DQN to that of R2D2, DQN required approximately 18 hours to complete 2.5M steps of training whereas each actor of R2D2 took approximately 4 hours to complete on the same hardware.

## 3.3 INTRODUCING DEEPER NETWORKS

Upon investigating possible improvements to be made to R2D2, I encountered existing research based on a variation of R2D2 termed *R2D4* (Recurrent Replay Deeper Denser Distributed DQN; Woolterton, 2021) which combined components of recent RL architectures such as *D2RL* (Sinha et al., 2020) and *Rainbow* (Hessel et al., 2017), which was implemented on the *Super Mario Bros.* environment (Kauten, 2018) found within OpenAI's gym. The results from this work provided me with the insight to take elements from D2RL and combine them with the model which R2D2 uses, as such additions had a noticeable improvement on the performance of R2D4. The main improvements listed in the paper are the use of a deeper network architecture with dense (skip) connections, along with some novel changes in using dropout.

D2RL allows for increased depth of feed forward neural networks by using skip connections from feature maps of previous layers to mitigate the instability in training due to issues such as vanishing gradients. The dense architecture addresses the decreasing sample efficiency of increasing layers in the multi-layered perceptron. Typically, performance is shown to decrease when increasing beyond two fully connected layers. This is likely due to the decrease in mutual information between the input and output because of the non-linear transformations used in deep learning (Sinha et al., 2020). Despite this, increasing layers can be valuable as they can enable improved optimisation for the networks. The deeper model consists of inputs concatenated with the hidden representations and fed into the next layer. This is modelled on all fully connected linear layers except the last output layer. The deeper R2D2 variants architecture shares the same convolutional body as DQN, followed by an LSTM cell which feeds into the Dueling head. The head has been modified to contain skip connections from the LSTM cell to the advantage and value streams at each additional layer with the intermediary layers requiring an input size of 1024 rather than 512, due to the concatenation of LSTM and linear layer outputs. The PyTorch pseudocode detailing the changes to the architecture can be found in the appendix (see Figure A.1) along with a basic graphical representation (see Figure A.2) and all relevant hyperparameters used.

# 4 RESULTS

The training process of the DQN agents were executed both locally and on the Hex cluster. This is because my DQN agent did not utilise the GPUs. Training times between the local machine and server were similar, requiring ~24 hours to complete 10M environment frames. On the other hand, R2D2 and 'Deeper R2D2' required the use of NVIDIA RTX 2080 graphics cards with 8GB RAM due to the actor-learner nature of the agents. The utilisation of GPUs led to a drastic performance increase which allowed the agents to perform twice as many environment steps in less than a third of the wall time. The DQN agents were trained and evaluated on all five of the chosen games whereas R2D2 and its deeper variant were only trained on Pong due to time constraints.

| **Easy Exploration** | | **Hard Exploration** | |
|---|---|---|---|
| Human-Optimal | Score Exploit | Dense Reward | Spare Reward |
| Breakout<br>Pong | Beam Rider | Q*Bert | Montezuma's Revenge |

Table 4.1: A taxonomy of the selected Atari 2600 games based on their exploration difficulty (Ostrovski et al., 2017).

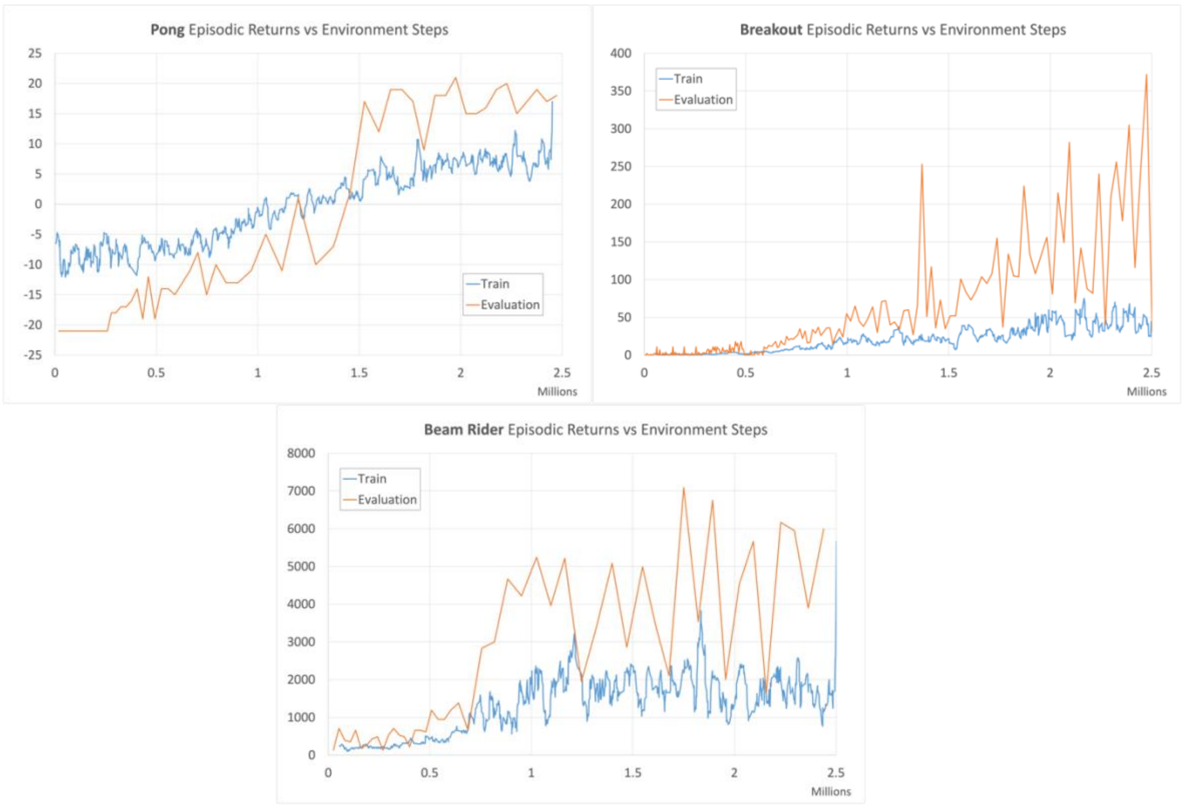


Figure 4.2: Episodic returns of training and evaluation runs on easy-exploration games: Pong (left), Breakout (right) and Beam Rider (bottom).

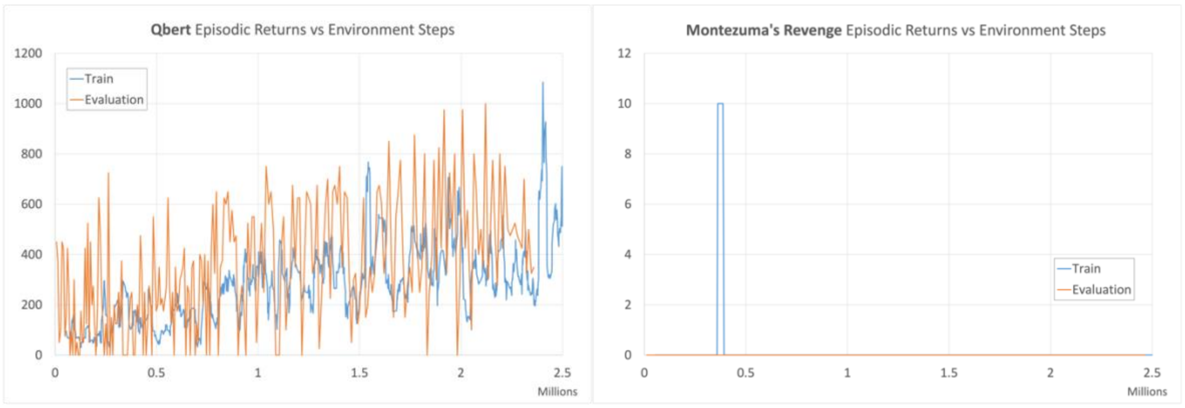


Figure 4.3: Episodic returns of training and evaluation runs on hard-exploration games: Qbert (left) and Montezuma's Revenge (right).

The evaluation agent rewards are typically higher because of the lower exploration rate at which a random move is played. The charts in Figure 4.2 demonstrate DQN's performance in easy-exploration games. It is evident that a DQN agent can effectively learn in these environments as the rewards are shown to gradually increase as the number of environment steps taken increase. Games such as Pong are solved fairly easily, requiring ~1.6M steps before the agent can consistently get a high ranging score. However, the performance in the other easy-exploration games are far from optimal when compared to average human scores. Observing the agent's rewards in Montezuma's Revenge (see Figure 4.3), it is clear that the agent struggles with sparse reward games. Although DQN may appear to perform better in dense reward games, the returns are subpar when measured against average human scores. This metric can be measured by calculating the Human Normalised Score $HNS = \frac{Agent_{Score} - Random_{Score}}{Human_{Score} - Random_{Score}}$. When evaluating the HNS, the number of training frames should be considered since it is about a quarter of the amount typically used in previous research. The agent scores used in this calculation are the final scores attained during the evaluation run.

| Game | Random Score | Human Score | DQN | Normalised DQN |
|---|---|---|---|---|
| Pong | -20.7 | 9.3 | 18 | 129% |
| Breakout | 1.7 | 31.8 | 380 | 1257% |
| Beam Rider | 363.9 | 5775 | 5990 | 104% |
| Qbert | 163.9 | 13455 | 800 | 5% |
| Montezuma's Revenge | 0 | 4367 | 0 | 0% |

Table 4.4: A table containing normalised DQN scores based on evaluation scores after 2.5M episodes of training. Random and human scores match those found within the DQN nature paper.

Upon inspecting the normalised values in Table 4.4, DQN performs relatively well in easy-exploration games when compared to the average human performance. On the other hand, it severely underperforms in hard-exploration games. R2D2 addresses some of the shortcomings of DQN and performs better in hard-exploration games. The results of DeepMind's R2D2 on the chosen games have been provided for reference and can be found in the appendix.

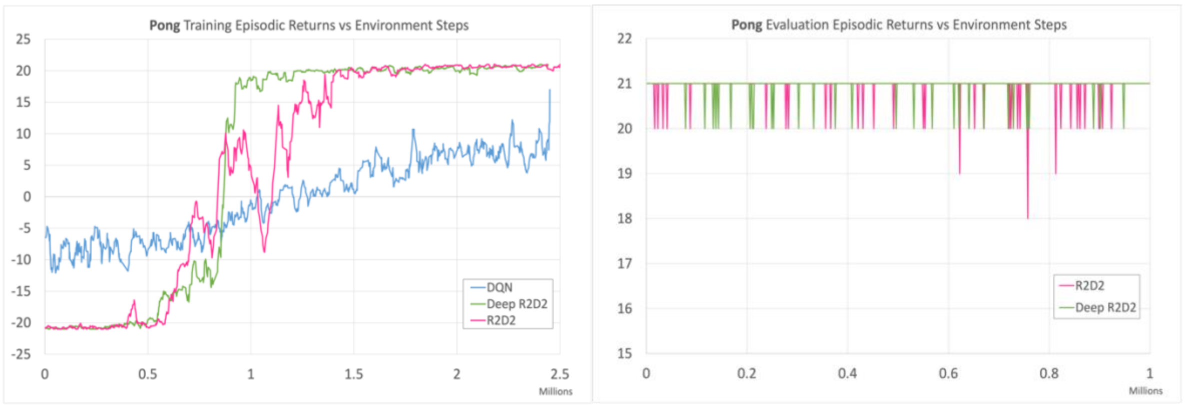


Figure 4.4: Training episodic returns of DQN, R2D2 and Deep R2D2 in Pong (Left). The training returns for R2D2 and Deep R2D2 are based on the highest scoring actors.

The training episodic returns of Deep R2D2 in Figure 4.4 illustrate how the proposed network architecture enables more stable learning. The learning curve of Deep R2D2 is considerably smoother than that of R2D2. Deep R2D2 is also shown to converge approximately 0.5M steps sooner than baseline R2D2, which supports research indicating that deep dense architectures can increase sample efficiency (Sinha et al., 2020). Additionally, the proposed variant performs better than the original in the evaluation run, consistently scoring either 20 or 21, unlike R2D2 which drops to 18. Both R2D2 and Deep R2D2 outmatch DQN with superior performance and converge in less than half of the number of steps required for DQN to converge.

## 4.1 EVALUATION

The proposed alterations to the model architecture used by R2D2 proved to be successful in improving the existing agent. To summarise, the updated model exceptionally increases the learning stability and sample efficiency of R2D2 on the tested game. This partially addresses my main objective to improve an existing algorithm to produce remarkable performance in an Atari game. The evaluation of this project is divided into the achieving following deliverables:

i. To implement DQN on a selection of Atari games and gain an understanding of the relevant software used to create and test RL agents in OpenAI Gym.
ii. To implement an advanced RL agent such as R2D2 and evaluate its performance in comparison to DQN across several games of different categories.
iii. To investigate areas in which an agent such as R2D2 can be improved and provide further insight to the RL agents in the Atari environment specifically.
iv. Demonstrate increased performance due to improvements implemented in R2D2.
v. Gain an advanced understanding of the mechanisms and design choices used to create an effective reinforcement learning agent.

By using these deliverables to assess the work completed, it would be fair to say that the project was a partial success. The main objective of improving R2D2 was accomplished, however a fair evaluation of Deep R2D2's performance across several games has not been delivered. An original implementation of DQN was coded from scratch to help understand the environments and software used. This also provided a greater understanding of the core components used in model-free RL agents. Despite the successes of achieving certain deliverables, the project was a failure in representing well-grounded evidence of improvements made to R2D2. This is primarily due to the fact that Pong isn't a good environment to assess the capabilities of a powerful agent since the scores are capped comparatively low and even simple agents can play Pong effectively. A more suitable environment to test R2D2 and Deep R2D2 on would have been Qbert, as it would demonstrate how the agents perform considerably better in a dense-reward game. The higher score 'caps' offered by games such as Qbert would allow for more useful insights to be drawn in comparison to games which are solved quickly. Such games would allow us to more easily distinguish superhuman performance from average human performance of RL agents. Training Deep R2D2 on Breakout would be ideal as it would allow us to measure and compare the sample efficiency of the agent on a solvable game with more frames required than the likes of Pong. Moreover, the project would have been a success given enough time and resources to be able to train the modified agent on the remaining chosen games. Unfortunately, I was not able code R2D2 from the ground up due to such constraints, so 3 of the 5 deliverables were achieved to a satisfactory level.

## 4.2 REMARKS

In hindsight, it would have been wiser to have setup a different cloud GPU service as a precaution. Not only would this have allowed me to simultaneously train multiple agents, but it also would have eliminated the memory issues that were suffered during the training processes. On multiple occasions, the processes would abruptly be terminated after many hours of training, which is primarily the reason I was unable to provide results for the remaining games. Over the course of this project, it became exceedingly laborious to monitor and restart each process when they were aborted. Other issues I had included being unable to upload files to the cloud service, and I would often have to wait hours before this was possible again. Although implementing DQN from start provided me with a better understanding of the mechanisms of a Deep RL agent, it proved to be time-consuming and ineffective towards the larger objective of this project. The time spent devising the code would have been better spent on further improving R2D2 with other novel contributions. Given the time and capability, I would also have tested Deep R2D2's performance across different hyperparameters in each game.

To address the limitations of this project, I would have altered the code to support multi-GPU systems as this would have solved the issues of having to resort to a single GPU with limited memory to train each agent. Furthermore, I would have investigated and performed the necessary changes to the code to support the more powerful NVIDIA GeForce RTX 3090 GPUs provided by Hex. Such changes would have effectively reduced the training times for each agent and have allowed me to produce more results. With that being said, I reiterate that the shortcomings of this project do not lie with the technology used throughout the project, but my inability to foresee potential issues that are expected to occur when undertaking research of this magnitude.

# 5 CONCLUSION

This project successfully investigates the combination of recent advances in reinforcement learning. By incorporating deep dense layers into the model architecture of R2D2, I have illustrated how the Deep R2D2 can provide for more stable learning and increase sample efficiency. Consequently, the new agent is shown to perform better in evaluation runs (albeit marginally due to the score cap of Pong). The work undertaken during the course of this project highlights some of many issues concerning reinforcement learning. One particular issue is in evaluating the performance of different agents trained on ALE and the necessity for a universal benchmark for a fair comparison. Furthermore, agent performance is typically compared to an ‘average’ human player which may not be the best indicator to assess how good an RL agent is at learning considering the number of games played. Instead, it would be more accurate to use a human world records baseline to assess the games on. Only then, would it be plausible for SOTA agents to undeniably claim superhuman performance. Many of these issues are handled by *SABER* (A Standardized Atari Benchmark for Reinforcement Learning; Toromanoff, Wirbel and Moutarde, 2019). In fact, some of the procedures found within this benchmark have translated to R2D2 such as the elimination of reward clipping and episode termination only when all lives have been depleted. By continuing these practices going forward, a more accurate picture of the gap between RL agents and peak human performance can be drawn.

## 5.1 FUTURE WORK

The results of this project leave much room for the investigation of the devised agent’s performance when hyperparameters are adjusted. More specifically, it would be interesting to see how the agent performs on increased batch and replay buffer sizes compared to that of baseline R2D2 and Agent57. Furthermore, the number of actors used can be investigated to determine if increasing this parameter can yield better rewards in comparison to that of R2D2 i.e., to measure the effect it may have on the new agent. Other areas in which this work can be further improved is by transforming the software to allow for multi-GPU training and add support for newer CUDA devices. Further research into model-based agents may also provide useful insights to add to this agent to increase its performance. Also addressing the areas of improvement relating to Agent57, enhancing the representations that SOTA model-free algorithms use for exploration is a possible avenue to take in further improving the agent.

## BIBLIOGRAPHY

Badia, A., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D. and Blundell, C., 2020. *Agent57: Outperforming the human Atari benchmark*. [online] Deepmind.com. Available at: <https://www.deepmind.com/blog/agent57-outperforming-the-human-atari-benchmark> [Accessed 3 April 2022].

Bellemare, M., Dabney, W. and Rowland, M., 2022. *Distributional Reinforcement Learning*. MIT Press.

Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M. and Dabney, W., 2020. Revisiting Fundamentals of Experience Replay. [online] Available at: <https://arxiv.org/abs/2007.06700> [Accessed 10 September 2022].

Gitau, C., 2018. *Success Stories of Reinforcement Learning*. [online] medium.com. Available at: <https://categitau.medium.com/success-stories-of-reinforcement-learning-9b4064171668> [Accessed 3 April 2022].

Guo, X., Singh, S., Lee, H., Lewis, R. and Wang, X., 2014. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. [online] Available at: <https://papers.nips.cc/paper/2014/hash/8bb88f80d334b1869781beb89f7b73be-Abstract.html> [Accessed 3 April 2022].

Hasselt, H., Guez, A. and Silver, D., 2015. *Deep Reinforcement Learning with Double Q-learning*. [online] Available at: <https://arxiv.org/pdf/1509.06461.pdf> [Accessed 9 May 2022].

He, F., Liu, Y., Schwing, A. and Peng, J., 2017. Learning To Play In A Day: Faster Deep Reinforcement Learning By Optimality Tightening. [online] Available at: <https://openreview.net/pdf?id=rJ8Je4clg> [Accessed 4 April 2022].

Hessel, M., Modayil, J., Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. and Silver, D., 2017. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. [online] Available at: <https://arxiv.org/pdf/1710.02298.pdf> [Accessed 9 May 2022].

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Hasselt, H. and Silver, D., 2018. *Distributed Prioritised Experience Replay*. [online] Available at: <https://arxiv.org/pdf/1803.00933.pdf> [Accessed 9 May 2022].

Hosu, I. and Rebedea, T., 2016. Playing Atari Games with Deep Reinforcement Learning and Human Checkpoint Replay. [online] Available at: <http://dmip.webs.upv.es/EGPAI2016/papers/EGPAI_2016_paper_7.pdf> [Accessed 4 April 2022].

Hu, M., 2022. *Deep RL Zoo: A collections of Deep RL algorithms implemented with PyTorch*. https://github.com/michaelnny/deep_rl_zoo.

Kaiser, Ł., Babaeizadeh, M., Miłos, P., Osinski, B., Campbell, R., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., Mohiuddin, A., Sepassi, R., Tucker, G. and Michalewski, H., 2020. Model Based Reinforcement Learning for Atari. [online] Available at: <https://openreview.net/pdf?id=S1xCPJHtDB> [Accessed 4 April 2022].

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R. and Dabney, W., 2019. *Recurrent Experience Replay In Distributed Reinforcement Learning*. [online] Available at: <https://openreview.net/pdf?id=r1lyTjAqYX> [Accessed 9 May 2022].

Kauten, C., 2018. *Super Mario Bros. for OpenAI Gym*. https://github.com/Kautenja/gym-super-mario-bros: GitHub.

Li, S., 2022. *Single-Machine Model Parallel Best Practices*. [online] PyTorch. Available at: <https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html> [Accessed 14 September 2022].

Machado, M., 2016. *The Success of DQN Explained by "Shallow" Reinforcement Learning*. [online] amii Research. Available at: <https://www.amii.ca/latest-from-amii/the-success-of-dqn-explained-by-shallow-reinforcement-learning/> [Accessed 9 May 2022].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. Human-level control through deep reinforcement learning. *Nature*, [online] 518(7540), pp.529-533. Available at: <https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf> [Accessed 10 September 2022].

Mnih, V., Koray, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2015. Playing Atari with Deep Reinforcement Learning. [online] Available at: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf> [Accessed 3 April 2022].

Moreno-Vera, F., 2019. Performing Deep Recurrent Double Q-Learning for Atari Games. [online] Available at: <https://arxiv.org/pdf/1908.06040.pdf> [Accessed 3 April 2022].

Ostrovski, G., Bellemare, M., Oord, A. and Munos, R., 2017. Count-Based Exploration with Neural Density Models. [online] Available at: <https://arxiv.org/abs/1703.01310> [Accessed 10 September 2022].

Samsami, M. and Alimadad, H., 2020. Distributed Deep Reinforcement Learning: An Overview. [online] Available at: <https://arxiv.org/pdf/2011.11012.pdf> [Accessed 10 September 2022].

Schaul, T., Quan, J., Antonoglou, I. and Silver, D., 2016. Prioritised Experience Replay. [online] Available at: <https://arxiv.org/pdf/1511.05952.pdf> [Accessed 4 April 2022].

Schrittweiser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhard, E., Hassabis, D., Graepel, T., Lillicrap, T. and Silver, D., 2020. *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. [online] Available at: <https://arxiv.org/pdf/1911.08265.pdf> [Accessed 9 May 2022].

Shao, K., Tang, Z., Zhu, Y., Li, N. and Zhao, D., 2019. A Survey of Deep Reinforcement Learning in Video Games. [online] p.1. Available at: <https://arxiv.org/pdf/1912.10944.pdf> [Accessed 3 April 2022].

Sharma, D., Prateek, M. and Sinha, A., 2013. Use of Reinforcement Learning as a Challenge: A Review. *International Journal of Computer Applications*, [online] 69(22), pp.28-34. Available at:

<https://www.researchgate.net/publication/272864640_Use_of_Reinforcement_Learning_as_a_Challenge_A_Review> [Accessed 3 April 2022].

Sinha, S., Bharadhwaj, H., Srinivas, A. and Garg, A., 2020. D2RL: Deep Dense Architectures in Reinforcement Learning. [online] Available at: <https://arxiv.org/abs/2010.09163> [Accessed 14 September 2022].

Stapelberg, B. and Malan, K., 2020. A survey of benchmarking frameworks for reinforcement learning. *South African Computer Journal*, [online] 32(2), p.266. Available at: <https://sacj.cs.uct.ac.za/index.php/sacj/article/view/746/413> [Accessed 3 April 2022].

Sutton, R. and Barto, A., 2015. *Reinforcement Learning: An Introduction*. London: The MIT Press.

Sutton, R., 2019. The Bitter Lesson. [online] Available at: <https://www.cs.utexas.edu/~eunsol/courses/data/bitter_lesson.pdf> [Accessed 10 September 2022].

Toromanoff, M., Wirbel, E. and Moutarde, F., 2019. *Is Deep Reinforcement Learning Really Superhuman on Atari? Leveling the playing field*. [online] Available at: <https://arxiv.org/pdf/1908.04683.pdf> [Accessed 9 May 2022].

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N., 2016. *Dueling Network Architectures for Deep Reinforcement Learning*. [online] Available at: <https://arxiv.org/pdf/1511.06581.pdf> [Accessed 9 May 2022].

Watkins, C. and Dayan, P., 1992. *Q-Learning*. [online] Available at: <http://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf> [Accessed 9 May 2022].

Woolterton, M., 2022. *Combining Recent Advances in Reinforcement Learning for Super Mario Bros.*. MSc. The University of Durham.

Zhang, S. and Sutton, R., 2018. A Deeper Look at Experience Replay. [online] Available at: <https://arxiv.org/pdf/1712.01275.pdf> [Accessed 10 September 2022].

# APPENDIX

## DEEP RL ZOO

The source code for the software used to train the R2D2 / Deep R2D2 agent can be accessed at: https://github.com/michaelnny/deep_rl_zoo. The changes made to the network architecture would be stored in ‘deep_rl_zoo-main/networks/dqn.py’. The repository has been forked to: https://github.com/emtiazsamad/deep_rl_zoo?organization=emtiazsamad&organization=emtiazsamad with the expected changes in place at the mentioned location. Other novel changes have been made to the parameters in ‘deep_rl_zoo-main/r2d2/run_atari.py’, such as the training and evaluation intervals which have been tailored to my needs. All credits for this software go the author, Michael Hu.

### Table A.1: DQN HYPERPARAMETERS

| **Hyperparameter** | **Value** |
|---|---|
| Batch Size | 32 |
| Replay Memory Size | 1000000 |
| Target Network Update Frequency | 1000 |
| Learning Rate (Adam) | 0.00025 |
| Discount Factor | 0.99 |
| Agent History Length | 4 |
| Exploration Rate | $1 \Rightarrow 0.1$ |
| Evaluation Exploration Rate | 0.005 |
| Optimizer | Adam |
| Loss Function | Huber |

### Table A.2: R2D2 / DEEP R2D2 HYPERPARAMETERS

| **Hyperparameter** | **Value** |
|---|---|
| Actors | 6 |
| Batch Size | 4 |
| Replay Memory Size (No. of Unrolls Stored) | 2500 |
| Target Network Update Frequency | 1500 |
| Actor Update Frequency | 400 |
| Unroll Length | 80 |
| Burn-in Length | 40 |
| Priority Exponent | 0.9 |
| Importance Sampling Exponent | 0.6 |
| *n*-step Returns (*n*) | 5 |
| Learning Rate (Adam) | 0.0001 |
| Discount Factor | 0.997 |
| Agent History Length | 4 |
| Exploration Rate | 0.001 |
| Evaluation Exploration Rate | 0.0001 |
| Optimizer | Adam |
| Loss Function | Huber |

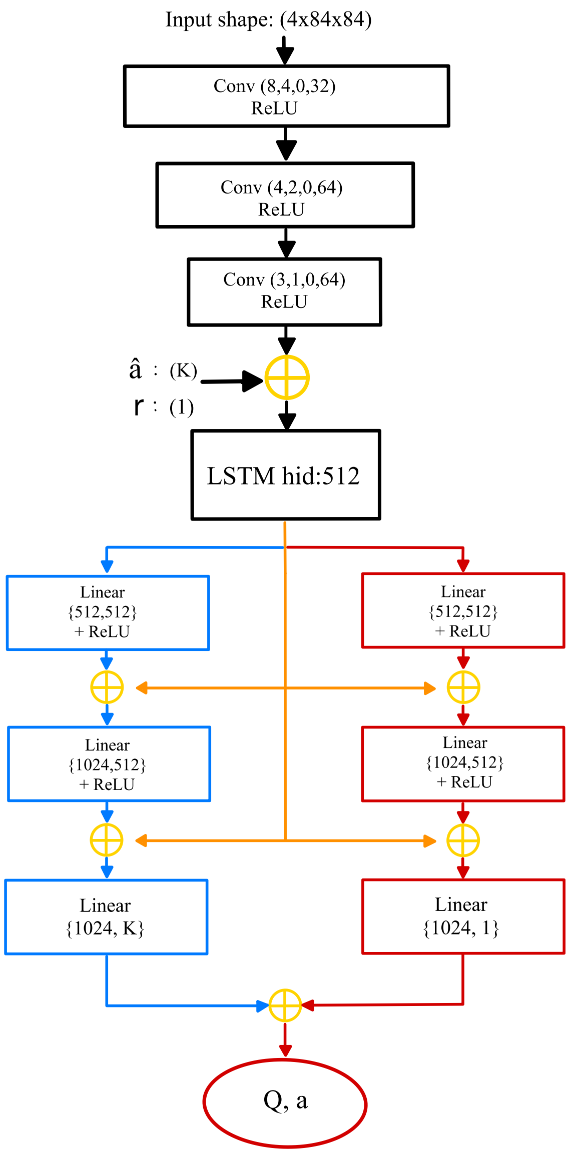
Figure A.1: PYTORCH PSEUDOCODE

The sections highlighted by the double lines indicate areas where the architecture has been adapted.

```
R2D2DqnConvNet:
        ...
        body = CNNBody(input_shape)
        core_output_size = body.output_size + num_actions + 1
        lstm = LSTM(input_size = core_output_size, hidden_size = 512, num_layers = 1)

        advantage_1 = Sequential(
                      Linear(lstm.hidden_size, 512),
                      ReLU(),
                      )
        advantage_2 = Sequential(
                      Linear(lstm.hidden_size * 2, 512),
                      ReLU(),
                      )
        ...
        advantage_4 = Linear(lstm.hidden_size * 2, num_actions)

        value_1 = Sequential(
                      Linear(lstm.hidden_size, 512),
                      ReLU(),
                      )
        value_2 = Sequential(
                      Linear(lstm.hidden_size * 2, 512),
                      ReLU(),
                      )
        …
        value_4 = Linear(lstm.hidden_size * 2, 1)

Forward:
        ...
        x = body()

        a1 = advantage_1(x)
        a1 = concatenate(a1, x)
        a2 = advantage_2(a1)
        a2 = concatenate(a2, x)
        ...
        a4 = advantage_4(a3)

        v1 = advantage_1(x)
        v1 = concatenate(v1, x)
        v2 = advantage_2(v1)
        v2 = concatenate(v2, x)
        ...
        v4 = advantage_4(v3)
```

## Figure A.2: BASIC REPRESENTATION OF MODEL ARCHITECTURE

It should be noted that this figure is a simpler recreation of the model architecture used in R2D4. As appropriate, full credits go to the author for the design of the figure (Woolterton, 2022). This diagram is purely provided for interpretability of the model architecture.


Input shape: (4x84x84)
Conv (8,4,0,32) ReLU
Conv (4,2,0,64) ReLU
Conv (3,1,0,64) ReLU
â : (K)
r : (1)
LSTM hid:512
Linear {512,512} + ReLU
Linear {512,512} + ReLU
Linear {1024,512} + ReLU
Linear {1024,512} + ReLU
Linear {1024, K}
Linear {1024, 1}
Q, a