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

School of Computer Science and Statistics

An Investigation into Deep Reinforcement Learning

Jack Cassidy
14320816

March 13, 2018

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of BAI (Computer Engineering)

Declaration

Abstract

A short summary of the problem investigated, the approach taken and the key findings. This should be around 400 words, or less.

This should be on a separate page.

Acknowledgements

Thanks Mum!

You should acknowledge any help that you have received (for example from technical staff), or input provided by, for example, a company.

Contents

List of Figures

List of Tables

Nomenclature

A	Area of the wing	m^2
B		
C	Roman letters first, with capitals...	
a	then lower case.	
b		
c		
Γ	Followed by Greek capitals...	
α	then lower case greek symbols.	
β		
ϵ		
TLA	Finally, three letter acronyms and other abbreviations arranged alphabetically	

If a parameter has a typical unit that is used throughout your report, then it should be included here on the right hand side.

If you have a very mathematical report, then you may wish to divide the nomenclature list into functions and variables, and then sub- and super-scripts.

Note that Roman mathematical symbols are typically in a serif font in italics.

1 Introduction

1.1 Motivation

Machine Learning (ML) and Artificial Intelligence (AI) in 2018 are subjects that are almost unique in their ability to permeate into nearly every sphere, community and space in today's society. From the research community to the business world and the public eye through extensive media coverage, ML is certainly becoming more and more of a de facto part of our everyday lives. Businesses employ recommender systems to suggest new products to their customers and predict the rise and fall of stock prices using function approximators like Deep Learning. Traditional home appliances are now outdated in favour of smarter, IoT systems that learn our habits and provide a more tailored experience.

ML is a broad umbrella term, encapsulating a variety of different approaches. Most ML tasks can be classified as either supervised or unsupervised learning. Deep Learning is fast becoming a popular and powerful technique in supervised learning, involving teaching artificial neural networks to approximate any function, given enough training data. Reinforcement Learning (RL), another subset of supervised learning, is a branch of ML that perhaps receives less public attention but is nonetheless believed to be set to revolutionize the field of AI Arulkumaran et al. (2017). Recent breakthroughs in the application of Deep Learning to RL algorithms has spawned the exciting research field of Deep Reinforcement Learning (DRL) which has produced to date unparalleled results in various AI domains, such as defeating the world champion Go player Silver et al. (2016).

There are a growing number of RL methods and algorithms, such as Monte-Carlo, Q-Learning, SARSA and Policy Search Sutton and Barto (1998). More recently, the advent of DRL has brought about adaptations to existing algorithms to expand their use to multi-dimensional observations spaces such as pixel information, a notable example being Deep Q-Learning Mnih et al. (2013). It is easy to become overwhelmed with all of these offerings when exploring the RL space. The motivation behind this project is to demystify the state of the art of RL.

1.2 Objectives

The objectives of this project are threefold.

1. Research the development and state of the art of RL.
2. Build a system to evaluate the performance of three state of the art DRL algorithms by collecting a series of metrics while applying each algorithm to a selection of Atari 2600 video games.
3. Carry out the experiments, obtaining values for game score, survival time and model loss. Compare and contrast the different algorithms using the metrics recorded.

The system is given no prior knowledge of how each game works and there is no change in the underlying architecture of the solution when applied to different games, all while maintaining a high level of performance. The aim for the system is to be a general solution, that it can be expanded to work for any number of games and algorithms in the future with ease of implementation. The algorithms used are Deep Q-Learning, Double Q-Learning and Dueling Q-Learning.

1.3 Research Methods

This project takes a *case study* based approach to the experimentation. The first phase of the project involves building the system to the specification outlined previously. The second phase treats each game entered into the system as an individual case study. The game ROM is given as input to the system. The game is simulated by a third party emulator of our choosing (discussed in chapter 3), from which the system extracts greyscale frames to learn from. The output of the system is an action that it has chosen to be optimal, selected from the discrete vector of possible actions as defined by the game's control scheme. This control scheme is not provided to the system, it determines it dynamically with each game. The action is fed back into the emulator and the cycle continues up to a terminating signal.

1.4 Report Overview

Chapter 2 gives some necessary background information. It will discuss the current state of the art of RL with particular interest in how it is being applied to video games, as well as the technologies and tools being used in research today and for this project.

Chapter 3 outlines the architecture of the system and the rationale behind certain design decisions.

Chapter 4 will discuss the components of the experiment evaluation. It will give a greater elaboration of the project's objectives, a description of the experimental setup, and a discussion of the results.

Chapter 5 closes the project with a conclusion of all that has been discussed, an outline of what has been achieved from both an objective and personal point of view and finally a suggestion for future work.

2 Background

This chapter will give an introduction to the fields of ML, RL and DRL. We will mainly focus on the knowledge that is pertinent to this project. The DRL section will outline the algorithms that are used in the comparisons later in the project. We will close with a discussion on the different tools used in RL research, specifically within the video game test bed.

2.1 Machine Learning

2.1.1 Introduction

ML is a broad umbrella term for various methods of giving computer systems the ability to 'learn' to complete some task efficiently using training and validation data, without being explicitly programmed to do so. Instead of following a programmed set of instructions to make a prediction, the ML system constructs a model that is a function approximated to some real world problem. ML tasks can be divided into two categories; supervised and unsupervised learning. In supervised learning, the dataset provides the correct output prediction, 'labelled' data, the system should make. The system can use the input/output pairs to iteratively learn the best prediction, using a combination of some generic error function, such as the *Mean Squared Error*, and the *Back Propagation* algorithm (Chauvin and Rumelhart (1995)) to update the model. In unsupervised learning, the dataset does not contain any output data points, 'unlabelled' data, hence it is more difficult to gauge the performance of an unsupervised ML algorithm. Unsupervised learning is generally used in the clustering of data into classes.

2.1.2 Development of Machine Learning in Video Games

In order to claim that an AI agent achieves general competency, it should be tested in a set of environments that provide a suitable amount of variation, are reflective of real world problems the agent might encounter, and that were created by an independent party to remove experimenter's bias (Bellemare et al. (2013)). In this way, video games provide an effective test-bed for efficiently studying general AI agents as they can provide all of these requirements. Although the application of ML generated AI to video games may seem novel, the end goal is not to produce agents for defeating world champion chess players, but to take these general agents and extend them to more pressing problems to humanity, of which there are endless possibilities.

The first application of notoriety to use computing to play a game arose in the research paper (Shannon (1950)), where mathematician Claude Shannon developed an autonomous chess-playing system. In that paper, author Shannon also highlighted the point that although the application of such a solution may seem unimportant;

> "It is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance"

Claude designed a strategy that, even at the time, was infeasible as it would take more than 16 minutes to make a move.

Fast forward to 1997, IBM developed "Deep Blue," a network of computers purpose built to play chess at an above-human level. It is renowned as the first AI system to defeat a world champion chess player, Garry Kasparov under normal game regulations. There is an air of controversy surrounding the feat, as IBM denied any chance of a replay after Kasparov claimed that IBM cheated by actively programming moves into Deep Blue as the game was in play.

In more recent times, British AI research company DeepMind published the paper (Mnih et al. (2013)) in which they achieved far and above human-level performance on a selection of 52 Atari 2600 games with their Deep Q-Network algorithm. DeepMind have since applied their techniques to modern, real time, strategy game StarCraft2 (Vinyals et al. (2017)), which we will discuss in the State of the Art chapter.

2.2 Reinforcement Learning

2.2.1 Introduction

RL is another case of ML tasks, which can come under the supervised and unsupervised learning categories. It is a general way of approaching optimisation problems by trial and error. An agent carries out actions in an environment, moving from one state to a new state and is given some positive or negative numerical reward. This is known as the *perception-action-learning loop*.

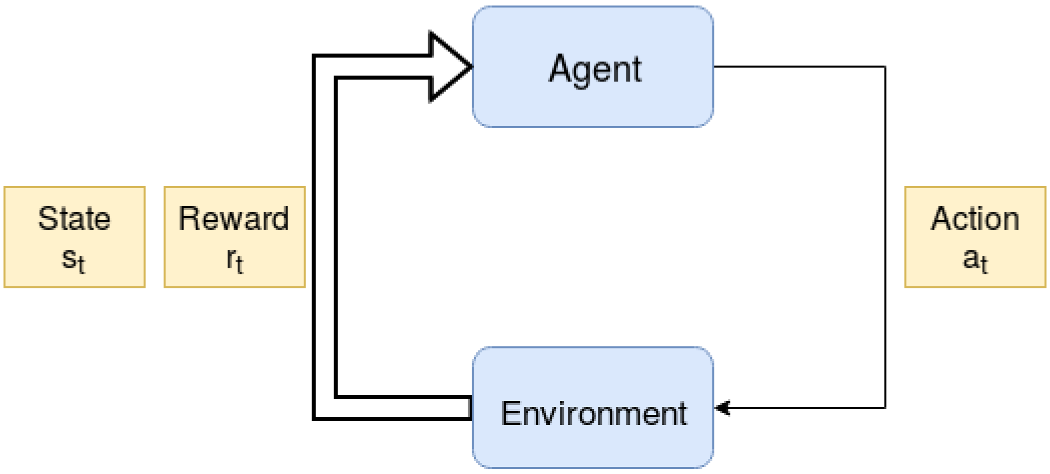

Figure 2.1: The perception-action-learning loop

RL is an interesting method of accomplishing ML tasks, as the agent can be given no prior information about it's environment or the task, and it can learn based solely on trial and error, reward and punishment. There are 3 main parts to a RL problem setup.

1. An agent follows a *policy* π, a rule that maps a state to an action.
2. A *reward function* $R(s, a)$, that gives an immediate value to an action a taken by the agent to transition from state s to s'
3. A *state value function* $V(s)$ that measures 'how good it is' to be in a given state. It assigns a value to the cumulate reward an agent can expect to gain by being in a state and following a policy through all subsequent states. We can define this as the *discounted cumulative reward*:

$$V(s) = E(\sum_{t=0} \gamma^t R(s_t, a_t)|s_0 = s) \qquad \forall s \in S \tag{1}$$

Where γ is a discount factor $[0, 1]$ and we choose $a_t = \pi(s_t)$. The objective of a RL problem is learning the optimal policy π^*, that for any given state will point the agent to the most favourable action so as to maximize it's cumulative reward, $V^*(s) = \max_\pi V(s)$. RL algorithms such as Q-Learning are used to find this optimal policy.

2.2.2 Markov Decision Process

In a more formal setting, it is a soft assumption that RL problems qualify as a Markov Decision Process (MDP) and can be modelled as such (Arulkumaran et al. (2017)). MDP's display the Markov Property; that the conditional probability distribution of future states is dependant only on the current state and totally independent of all past states. A MDP consists of:

- A finite set of states S
- A finite set of actions A
- A transition function $P(s, a, s') = P(s_{t+1}|s_t, a_t)$, a model mapping current state, action pairs to a probability distribution of potential future states.
- An immediate reward function $R(s, a)$

Again, an MDP seeks to find the optimal policy π^*. We define π^* as

$$\pi^* = argmax_a\{\sum_{s'} P(s, a, s')(R(s, a) + \gamma V(s')\} \tag{2}$$

2.2.3 Model-Free Learning

Not all RL problems are provided with a transition function $P(s, a, s')$. In fact, it is more often than not that we cannot express the agent's environment with a model. Such a scenario is called *model-free learning*, where the agent must learn the optimal policy without the use of a transition function to guide it on which action to take. Instead it must devise some other way of modelling it's environment, such as building a 'memory' of actions and rewards based on experiences and deriving an optimal policy from these experiences. The downside to this is the potentially large amounts of auxiliary space needed to store the experiences. This is where algorithms such as Q-Learning are used.

Q-Learning is a model free RL algorithm. At each state, the agent calculates an immediate reward, based solely on the current state and action taken, and the *quality* of taking an action a in state s and following a policy π thereafter, called the Q-Value, which is defined as:

$$Q(s,a) = E(\sum_{t=0} \gamma^t R(s_t, a_t) | s_0 = s, a_0 = a) \qquad \forall s \in S \tag{3}$$

Where we have chosen a_0 arbitrarily and choose all subsequent $a_t = \pi(s_t)$ thereafter. As the agent explores all states multiple times and experiments with different actions, the corresponding Q-values are saved and updated in a data structure, hence an optimum policy can be derived by finding the optimum Q-values $Q^*(s,a)$ for all states after a predetermined number of iterations or until the policy is 'good enough'. The update step for a Q-Value is defined as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{4}$$

$$\delta = r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{5}$$

We define δ as the Temporal Difference Error (TDE). It is an error function that is used in many RL algorithms. α is a hyperparameter called the *learning rate* chosen in the range $(0,1)$. As $t \to \infty$, $Q_t \to Q_t^*$, we converge on an optimum solution. The data structure used to store the agent's experiences can be referred to as the Q-Matrix. It is a $|S|x|A|$ sized matrix, that is the total number of states x total number of available actions.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

Figure 2.2: An example Q-Matrix. 0 indicates an unexplored state, action pair

The agent follows algorithm 1, detailed below. After a suitable number of iterations and exploration, the Q-Matrix becomes a 'map' for the agent, whereby it can look up the action with the highest Q-Value for any state (Watkins and Dayan (1992)). Q-Learning is a straight-forward, elegant solution to a RL task. However, for an environment space of high dimensionality, such as an array of RGB pixels from an image, the Q-Matrix becomes infeasibly large in the S dimension and increasingly sparse, as only a small percentage of the total available state, action pairs will be visited. As a worked example, imagine a robot that is using Q-Learning to find a path from it's current position to some exit room. If the robot takes 210x160, 8-bit colour space, RGB photos of it's surroundings to represent a state, the S dimension becomes $256^{210\text{x}160\text{x}3}$ in size. A solution to the dimensionality problem was proposed by DeepMind, in the paper (Mnih et al. (2013)) which will be discussed later in this chapter.

Algorithm 1 Q-Learning Algorithm

```
1: procedure BUILDING Q-MATRIX
2:     Set $\alpha$ and $\gamma$ parameters.
3:     Initialize Q-Matrix to zero.
4:     repeat
5:         while Goal/terminal state not reached do
6:             Select $a$ randomly from all possible actions in current state
7:             Consider going to state $s_{t+1}$ from state $s$ using action $a$
8:             Get maximum Q-Value from $s_{t+1}$ considering all possible actions
9:             $Q(s,a) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1},a))$
10:        end while
11:    until Policy good enough
12: end procedure
13: procedure USING Q-MATRIX
14:     $s \leftarrow$ initial state
15:     while Goal/terminal state not reached do
16:         $a \leftarrow max_a Q(s,a)$
17:         $s \leftarrow s_{t+1}$ taking action $a$
18:     end while
19: end procedure
```

2.2.4 Exploration vs. Exploitation

There is a fundamental issue in any RL task, where the agent must choose between taking an instantaneous reward by exploiting the policy, or taking a random action to explore the environment in search of a potentially higher long term reward. This problem is illustrated by a well-known RL problem known as the k-armed bandit problem.

> "The agent is in a room with a collection of k gambling machines (each called a 'one armed bandit' in colloquial English). The agent is permitted a fixed number of pulls, h. Any arm may be pulled on each turn. The machines do not require a deposit to play; the only cost is in wasting a pull playing a suboptimal machine. When arm i is pulled, machine i pays off 1 or 0, according to some underlying probability parameter p_i, where payoffs are independent events and

the p_i's are unknown. What should the agent's strategy be?" (Kaelbling et al. (1996))

The amount of time the agent spends in the environment is one factor that can be taken into consideration when making this decision. In general, the longer the agent spends in the environment, the less impact taking an exploratory approach, sometimes towards a sub-optimal policy, will have on the end policy.

One solution to this dilemma is to take an *epsilon greedy policy*. At each state, the agent takes a random action with a probability of ϵ and an action from the policy with a probability of $(1 - \epsilon)$. ϵ is linearly reduced at each iteration to some predetermined floor value. This way, the agent will spend more time exploring at the start of it's interaction with the environment, and will then (hopefully) converge to an optimal policy as it progresses.

2.3 Deep Reinforcement Learning

2.3.1 Introduction

DRL refers to the use of deep learning algorithms within the field of RL. As mentioned previously, RL struggles with environments of high dimensionality. DRL overcomes this issue thanks to the universal function approximation property of deep neural networks and their abilities to isolate and recognize features of interest within high dimensional data and to compactly represent that high dimensional input data (Arulkumaran et al. (2017)). DRL can utilize a convolutional neural network to learn a representation of the environment on behalf of the agent through high dimensional sensory input such as video data. A set of fully connected layers are then generally used to approximate the target of the underlying RL algorithm, such as $V(s, a)$, $Q(s, a)$, an action etc. The deep neural network can then be trained using an appropriate variant of the backpropagation algorithm, such as stochastic or batch gradient descent.

2.3.2 Deep Q-Network

The event that brought DRL to the attention of the research community was from the paper (Mnih et al. (2013)). DeepMind created a variant of Q-Learning called Deep Q-Network, that achieved above-human level performance on a large selection of 52 Atari 2600 video games. They combined a convolutional neural network for feature detection, and a fully connected network to learn the Q-Values for all available actions, with ReLU activations within each layer. The network uses a standard Least Square Error loss function in training with gradient descent, defined as:

$$L = (y_t - Q(s, a))^2 \tag{6}$$

$$y_t = r + \gamma max_a Q(s_{t+1}, a) \tag{7}$$

This architecture was named the Deep Q-network. This breakthrough successfully removes the dimensionality problem, as there is no need to keep a data structure storing all previous experiences. The deep neural network takes an array of RGB pixel information, taken as a stack of 3/4 (depending on the game) greyscale frames, as input to the convolutional network. The fully connected network outputs a vector of Q-Values for each available action in the game.

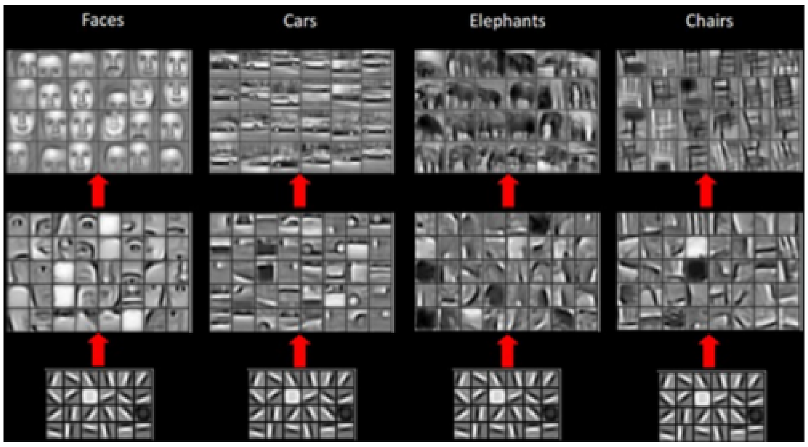

Figure 2.3: Examples of output filters from a convolution neural network

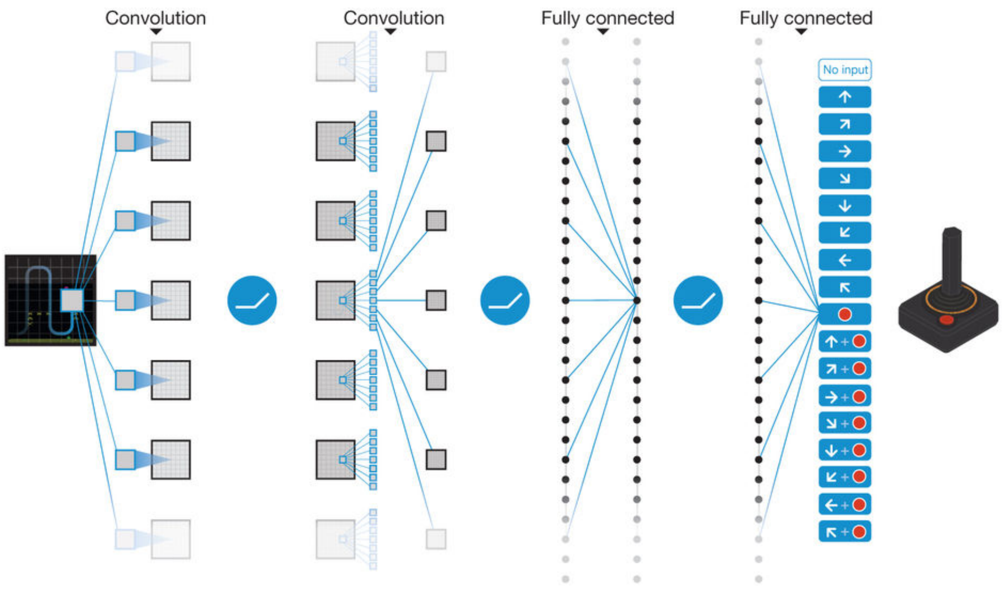

Figure 2.4: The original Deep Q-network Architecture

2.4 RL Research Tools for the Video Game Test Bed

2.4.1 OpenAI Gym

OpenAI Gym is a high-level Python API that provides a suite of environments on which to perform RL research (Brockman et al. (2016)). These environments range from physics problems such as balancing a pole, to a small selection of Atari 2600 video games, to robotics tasks like teaching a robot to walk or landing a space ship on a planet safely.

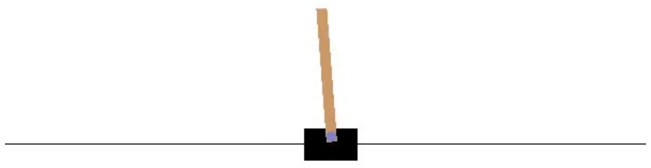

Figure 2.5: The CartPole environment, where an agent must move the cart left and right to keep the pole balanced

```
        state = env.reset() # create starting state
        action = agent.act(state) # agent chooses first action

        nxt_state, reward, done, info = env.step(action)
        if done:
            break
        ...

```

Figure 2.6: Typical code snippet from OpenAI Gym Python API

The user must write their own agents to interact with the provided environments. In the above code snippet, the `env` object is one of the environments provided by the API, and the `agent` object would be written by the user, including the `act(state)` method. The `(nxt_state, reward, done, info)` tuple is about the extent of the information about the environment that can be gathered using the API. For this reason, as will be discussed in the Design chapter, OpenAI Gym was not used in the implementation of this project. However, it remains an excellent tool for a beginner to become acquainted with RL problems.

2.4.2 The Arcade Learning Environment

The Arcade Learning Environment (ALE) (Bellemare et al. (2013)) is a framework for assisting RL researchers in testing AI agents on Atari 2600 video games. It is similar to OpenAI Gym, in fact OpenAI Gym uses ALE under the hood for it's Atari 2600 environments, however it is much more low level. It's main supported language is C++, it will provide full functionality to any agents written in C++. There is however an excellent Python interface with almost complete functionality; more than enough for the purposes of this project. For languages other than C++ or Python there is an intelligent text based mechanism using pipes called the FIFO Interface that allows *any* programming language to use ALE with a restricted set of services.

ALE provides much more functionality than OpenAI Gym without being much more complex to program with. We can load a selection of 52 supported Atari 2600 ROMs to experiment with, all having different graphics, control schemes, scoring mechanisms etc. hence ALE provides a nicely varied selection of environments against which to test AI agents. Just some of the features we can extract while an agent is running are:

- The current game screen as an RGB or greyscale array of pixels.
- The number of frames the agent has played in total and since the last game reset.

- A list of all available actions on the Atari 2600 and a tailored list of actions used in the specific game being played.
- The number of lives (if provided) the agent has remaining.
- Query at any time if the game has terminated (game over), not just after a state transition as in OpenAI Gym
- Change difficulty mode, if the specified game supported it on the original Atari 2600.
- Video and sound recording
- The internal state of the 128-byte RAM of the game.
- Load and save states.

As well as all of the expected features such as processing an action in the environment and returning a reward. By default, the returned reward is the difference in game score during the state transition caused by an action.

ALE provides many more practical features than OpenAI Gym, however there are some drawbacks to using ALE. The researcher is restricted to using just Atari 2600 environments, which means an inherently high-dimensional problem as the agent will more than likely represent a state as an array of pixels. OpenAI Gym provides a lot more lower-dimensional environments, such as the physics based CartPole where the state is represented as a much shorter tuple of `(cart position, cart velocity, pole angle, pole velocity)`.

3 State of the Art

The current state of the field of RL is vast. More and more it is being utilized as the next step towards more efficient and intelligent ML systems (Arulkumaran et al. (2017)). This section will be divided into two discussions. The first will give a basic survey of the types of different applications of RL today. The second section will discuss the state of the art for RL research within the video game test bed.

3.1 Robotics

Robotics is a field very suited to RL techniques. It closely follows the basic definition of an RL task involving an agent exploring an environment through trial and error to maximize some reward function. Although it is well suited, there are many challenges that face RL techniques in robotics. The recent work of (Mnih et al. (2016)), the A3C algorithm (see subsequent section for an explanation of the algorithm), is an example of state of the art RL techniques addressing one of the most pressing issues for RL in robotics: the 'curse of dimensionality' with respect to the navigation of a robot through a complex environment. Coincidentally A3C has provided state of the art results in video game AI agent research results.

The representation of state in robotics is often imperfect and noisy due to it's high levels of dimensionality and the resolution reduction of continuous physical measurements to discrete digital computer/controller representations. A mechanical representation of state such as robot limb position, or any other physical measurement can have many degrees of freedom (DOF). In addition to state complexity, the action space in contrast to the Atari domain which is fixed and simple, can reach high dimensionality due to multiple DOF and the hardware complexity of most modern robots, where there can be a large number of motors and other components working in tandem, each adding to the complexity of the total action space.

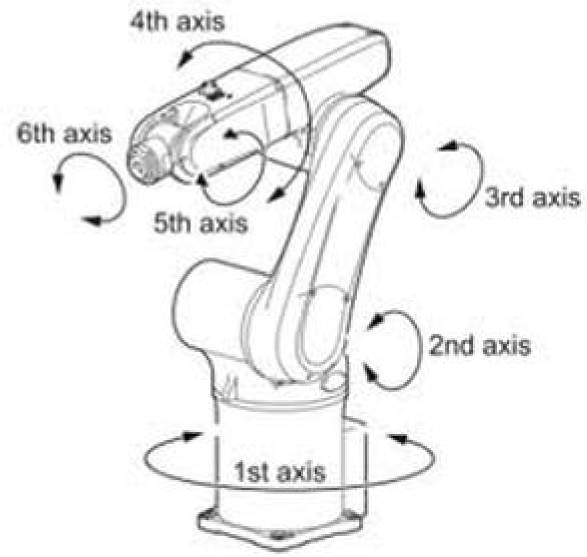

Figure 3.1: Example DOF figure of a robotic arm

With respect to a robot that navigates through a complex, partially observable environment with dynamic obstacles, in addition to the high dimensional state size, the memory efficiency of remembering past experiences becomes a pressing issue as it is of utmost importance for a robot learning to navigate to be able to utilize past experiences to map its surroundings. A3C abandons experience replay in favour of a stacked Long Short Term Memory (LSTM) network running multiple agents that are exploring the environment in parallel, providing a wider range of experiences that scales with the number of parallel agents employed. This dramatically increased the memory efficiency of the implementation.

3.2 Natural Language Processing

Deep learning, more so than DRL, has been permeating into the field of Natural Language Processing (NLP). There have been many exciting revelations in NLP in recent years that interface with users regularly, such as chat bots, text prediction and machine translation (MT) etc.

MT is simply defined as the sub-field of computer linguistics that investigates the use of software to translate text or speech from one language to another. The ability for systems to recognize full phrases and sentences and translate those accurately instead of simply substituting words is what defines good MT systems from others, as the latter will almost always be rife with grammatical errors in the target language. Neural Machine Translation (NMT) introduces deep learning into the MT sphere, to further the ability to recognize phrases and sentences. Recurrent neural networks (RNNs) are popular in NTM systems, owing to their internal memory which is suited to picking out patterns and phrases, as opposed to fully feed forward networks with no concept of internal remembrance. The basic architecture involves two RNN's, an encoding layer at the input sentence and a decoding layer at the translated sentence output.

The current state of the art in NMT is Google's Neural Machine Translation (GNMT) system Wu et al. (2016). It tackles the issues of speed of translation/inference, robustness in the face of translating sentences containing 'rare' words and incomplete translations. These are three issues that have *severely* hampered the deployment of MT systems into production. The network architecture consists of an LSTM encoder with 8 layers and the same at the output. The residual connections between layers in the encoder and decoder, to introduce parallelism, and low-precision arithmetic during inference computations are used to tackle the speed of translation problem. Breaking up words into sub-word units or 'wordpieces' reduces the chances of finding 'rare' words by translating on sections rather than full words, but provides a balance of compute time over single character scale translation. Finally, a beam search technique to tackle the issue of incomplete translations. With these three features, GNMT has proved itself as a state of the art for MT and is in fact used in production at Google today.

3.3 Deep Reinforcement Learning

3.3.1 Improvements to Deep Q-Network

In 2013, the DQN breakthrough was the state of the art. Since then, many improvements and adaptations have been made on this model, by DeepMind and other research groups. Such examples include the Double Q-Network adaptation (Van Hasselt et al. (2016)) and the Dueling Q-network architecture (Wang et al. (2015)). Experience replay and target networks were two techniques added to the original architecture by DeepMind to add more stability to the learning process. Prioritized experience replay was then built on from experienced replay providing even better performance.

Experience Replay and Target Network

Instead of learning from the immediately previous experience when training the network, a large memory of past experiences are stored as tuples of $(s_t, a_t, r_t, s_{t+1}, term)$, where *term* is a boolean indicating if this state transition was terminal (a gameover). The network is then trained from a random sampling of past experiences from this replay memory. It was found to greatly reduce the number of interactions the agent needed to have with the environment. However, this technique *is somewhat* limited as there is no way to differentiate important experiences from unimportant ones.

The target network is a secondary network $\hat{Q}$ cloned from the online Q-network Q, that is used to predict the targets y_i when training Q. The weights of $\hat{Q}$ are cloned from Q every τ training steps. This modification makes the algorithm more stable, as an increase to $Q(s_t, a_t)$ was often found to also increase $Q(s_{t+1}, a)$ for all a, thus also increasing the targets y_i. This can create a diverging solution in some cases. Freezing the weights makes the updates to Q and the targets y further apart, decreasing the likelihood of divergence (Mnih et al. (2015)).

Algorithm 2 Deep Q-Network Algorithm with Experience Replay and a Target Network

```
1: Initialize replay memory $D$ to capacity $N$
2: Initialize $Q$ with random weights $\theta$
3: Initialize $\hat{Q}$ with weights $\theta^-$ cloned from $\theta$
4: for episode = 1, M do
5:     Initialize arbitrary first sequence of frames for initial state
6:     for t = 1, T do
7:         With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = max_a Q(s_t, a)$
8:         Execute action $a_t$ and observe reward $r_t$ and state $s_{t+1}$
9:         Store state transition $(s_t, a_t, r_t, s_{t+1}, term)$ in $D$
10:        Sample random mini batch $(s_i, a_i, r_i, s_{i+1}, term)$ from $D$
11:        if $term = true$ then
12:            Set $y_i = r_i$
13:        else
14:            Set $y_i = r_i + \gamma max_a Q(s_{i+1}, a)$
15:        end if
16:    end for
17:    Perform a gradient descent step on $(y_i - Q(s_i, a_i))^2$
18:    if $t$ is a multiple of $\tau$ then $\theta^- \leftarrow \theta$
19:    end if
20: end for
```

Double Deep Q-Network

In the standard Deep Q-Network algorithm 2.14, when training we use the same Q-Value to select and evaluate an action for the target y. This can result in overoptimistic Q-value estimates over time, leading to sub-optimal policies as certain actions are erroneously favoured over others due to early over estimations. Double Deep Q-Network (Van Hasselt et al. (2016)) separates action selection from action evaluation in the target y. The online network Q still estimates the best action to take based on a max operator on it's predicted vector of Q-values. We reuse the target network to then evaluate the effectiveness of this action. The updated target equation is given as:

$$y_i = r + \gamma \hat{Q}(s_{i+1}, argmax_a Q(s_{i+1}, a)) \tag{1}$$

As with the original target network optimisation, the weights from Q are copied to $\hat{Q}$ every τ training updates.

Algorithm 3 Double Deep Q-Network Algorithm

```
 1: Initialize replay memory $D$ to capacity $N$
 2: Initialize $Q$ with random weights $\theta$
 3: Initialize $\hat{Q}$ with weights $\theta^-$ cloned from $\theta$
 4: for episode = 1, M do
 5:     Initialize arbitrary first sequence of frames for initial state
 6:     for t = 1, T do
 7:         With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = max_a Q(s_t, a)$
 8:         Execute action $a_t$ and observe reward $r_t$ and state $s_{t+1}$
 9:         Store state transition $(s_t, a_t, r_t, s_{t+1}, term)$ in $D$
10:         Sample random mini batch $(s_i, a_i, r_i, s_{i+1}, term)$ from $D$
11:         if $term = true$ then
12:             Set $y_i = r_i$
13:         else
14:             Set $y_i = r_i + \gamma \hat{Q}(s_{i+1}, argmax_a Q(s_{i+1}, a))$
15:         end if
16:     end for
17:     Perform a gradient descent step on $(y_i - Q(s_i, a_i))^2$
18:     if $t$ is a multiple of $\tau$ then $\theta^- \leftarrow \theta$
19:     end if
20: end for
```

3.3.2 Current State of the Art for Deep Q-Network

Dueling Q-Network Architecture

The Dueling Q-Network Architecture is an improvement on the original Deep Q-Network's single stream, convolutional network into a single fully connected layer network architecture. It *does not* provide any change to the underlying algorithms at work. For this reason, it can be applied to other RL algorithms that use Q-Values. The single stream fully connected layer is separated into two streams. One stream estimates the state value functions $V(s)$ and the other estimates a new function, the action advantage function $A(s, a)$.

$$A(s, a) = Q(s, a) - V(s) \tag{2}$$

The two streams are aggregated at the final layer to output $Q(s, a)$. The convolutional network in the upper half remains unchanged. $V(s)$ and $Q(s, a)$ we have explained the intuition for previously. The action advantage function $A(s, a)$ gives a relative importance of each action in a given state. Hence, the dueling architecture can learn which states are valuable separately from the effect of each action in each state.

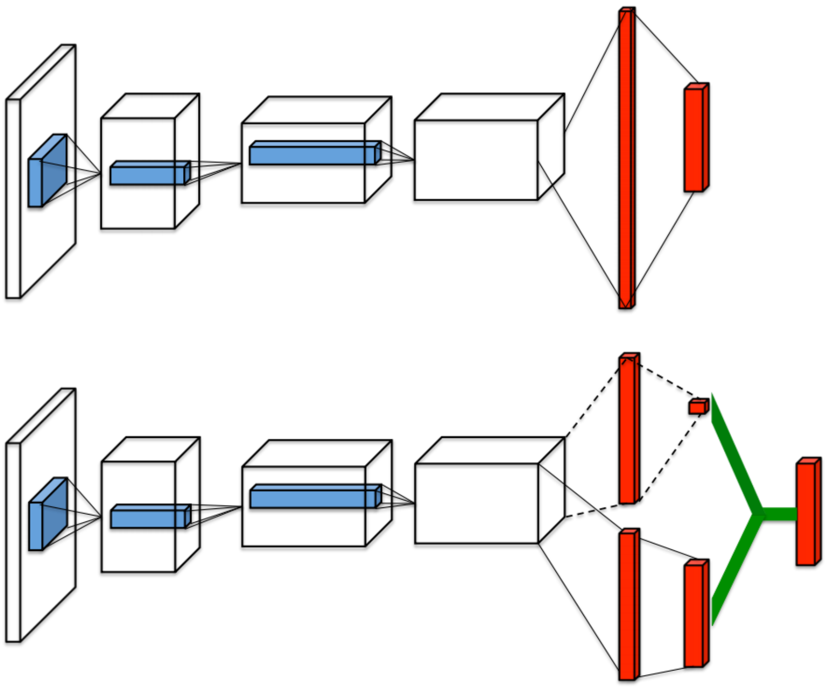

Figure 3.2: The original Deep Q-Network (above) vs. Dueling Q-Network (below) architecture

The aggregating layer is not a simple sum of $A(s, a)$ and $V(s)$. It was found that equation 2 "is unidentifiable in the sense that given Q we cannot recover V and A uniquely" (Wang et al. (2015)). Instead we use a slightly augmented version:

$$Q(s, a) = V(s, a) + (A(s, a) - max_{a'} A(s, a')) \tag{3}$$

Prioritized Experience Replay

In regular experience replay, transitions are sampled randomly and uniformly from the replay memory collection, with no regard to which transitions the agent might learn more from at any given time. Prioritized experience replay was first suggested by (Schaul et al. (2015)) as a way to improve the efficiency of regular experience replay by replaying more important transitions more frequently. It is difficult to quantify the importance, or potential learning progress an agent can expect by replaying a transition, hence a reasonable estimation can be made to approximate it. (Schaul et al. (2015)) approximate learning progress with the magnitude of the TDE of a transition, a suitable value as many RL algorithms use TDE, including Q-Learning. Along with the tuple $(s_t, a_t, r_t, s_{t+1}, term)$, the TDE, δ is now also stored with each state transition. The TDE of each transition is updated after being sampled to prevent the highest rank transition from always being sampled.

A greedy prioritization, where the maximum TDE experiences are always sampled is flawed. Transitions that first occur with a low TDE will effectively never be revisited. The same small set of experiences are repeatedly sampled, with such a lack of diversity the model is likely to overfit. To overcome this (Schaul et al. (2015)) propose a stochastic prioritization policy with two variants.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{4}$$

$$p(i) = |\delta_i| + \epsilon \tag{5}$$

$$p(i) = \frac{1}{rank(i)} \tag{6}$$

Where $P(i)$ is the probability of sampling transition i, k is the number of transitions in the replay memory, $p(i)$ is the priority of transition i and it's two variants are shown. Variant 1 is similar to a greedy prioritization, but ϵ prevents 0 TDE transitions from never being sampled. Variant 2 is based on *rank*, where $rank(i)$ is the rank of transition i over all transitions when sorted by $|\delta|$.

3.4 Video Games

3.4.1 A3C

The team at DeepMind, in (Mnih et al. (2016)) propose a new framework for training DRL agents. Multiple copies of the same agent are run asynchronously on a separate thread, all performing gradient descent updates to a single common neural network. At any given time, every agent will likely be experiencing a different state, giving a much greater view of the environment. This helps to mitigate the exploration vs. exploitation dilemma previously discussed. The framework, which we will refer to as 'asynchronous methods,' can be used to augment many different pre-existing DRL algorithms to improve their performance. As well as performance improvements, this framework significantly cuts the time to train an agent in the Atari 2600 test bed and can be accomplished on a standard multi-core CPU, as opposed to previous methods that have used specialized, expensive, GPU hardware. The results obtained in (Mnih et al. (2016)) from their experiments were carried out on an unspecified 16-core CPU and an Nvidia K40 GPU. At the time of writing, an AMD Ryzen 1950X 16-core CPU costs $874 and an Nvidia K40 costs $2300 (prices from `www.amazon.com`). Thus asynchronous methods further lowers the economic barrier to entry to DRL.

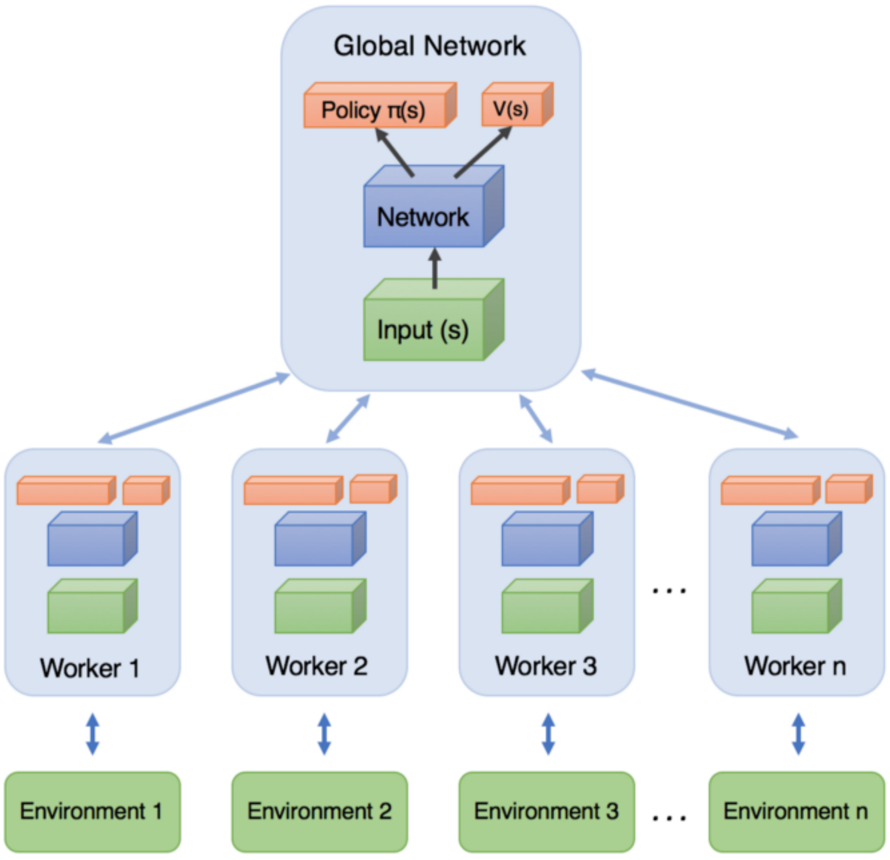

Figure 3.3: High level overview of Asynchronous Methods

Of the four augmented algorithms tested in (Mnih et al. (2016)), Asynchronous Advantage Actor-Critic (A3C) stands as the current state of the art algorithm for training DRL agents. It performs better than all other competitors, including the most recent revisions of Deep Q-Network, in the Atari 2600 test bed in terms of game score, and in half the time of the previous state of the art. Additionally, DeepMind applied A3C to an interesting problem outside of the video game domain. An AI agent was taught to navigate the 3D maze environment Labyrinth, collecting rewards. The collectables were apples - worth 1 point and portals - worth 10 points. The agent was placed in a randomly generated maze each episode and was given 60 seconds to accumulate as much reward as possible. The agent peaked with an average score of 50, indicating that it had learned a general strategy for exploring randomised environments. The applications of this feat to fields such as robotics is exciting.

3.4.2 Games of Imperfect Information

Games of imperfect information mean that the player is unaware of the state or actions chosen by the opposite player. Games of imperfect information include Texas Hold'Em Poker, as players do not know the other player's card and Real-time strategy (RTS) games, as players cannot see the areas of the map where the other players are until they explore those areas. These scenarios provide new challenges for training more intelligent AI agents.

Case Study: StarCraft 2 Learning Environment

The StarCraft 2 Learning Environment (SC2LE) is a platform for testing RL AI agents in the game StarCraft 2, an RTS of imperfect information (Vinyals et al. (2017)). It was created by DeepMind in collaboration with Blizzard Entertainment, the creators of StarCraft 2. The game is exceedingly more complex than the Atari 2600 games we have discussed previously.

- Modern graphics. StarCraft 2 is a modern 3D game with a movable isometric camera perspective. This makes representing observations as a stack of game frames very difficult, as the pixel counts are high and parallax error is introduced.
- Action space. The diversity of actions in StarCraft 2 is far higher, with approximately 10^8 possibilities in a point-and-click fashion. Many actions require a sequence of primitive actions such as, drag box around units, select building to build, place building on map. The player can be controlling potentially hundreds of units of many different types and abilities, as well as having to manage resources, building and ensure the opposing players aren't attacking.
- Multiple agents. There can be up to 4 players in one game, all competing against one another for map control. This is a stark contrast to the single player Atari 2600 games competing against a single built-in game AI.

- Imperfect information. There is a 'fog-of-war' element that hides parts of the map the player does not actively have units in.
- Delayed rewards. The action of building a number of units, exploring an area of the map etc. can have rewards that do not materialize for many, many time steps. This provides a new frontier for agents capable of creating long term strategy.

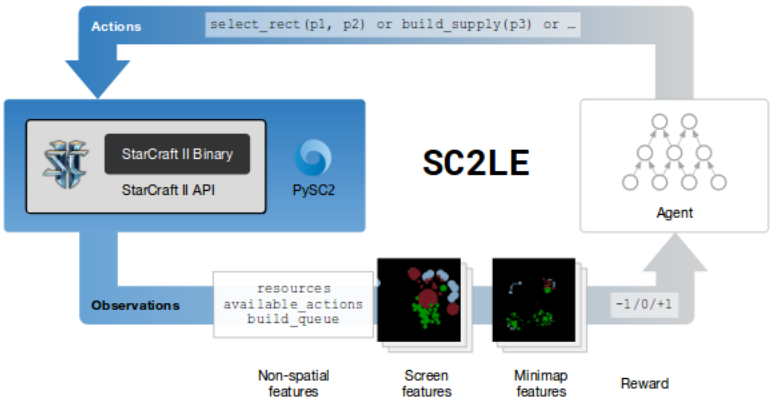

Figure 3.4: SC2LE interacting with an AI agent

For a more complete breakdown of the complexity of StarCraft 2, we refer the reader to the paper (Vinyals et al. (2017)), specifically section 3.2.

SC2LE provides a high level Python API for programmatic interaction with StarCraft 2, called PySC2 that has been optimized for training RL agents. To tackle the aforementioned problem of high dimensional 3D game graphics, PySC2 abstracts away the game's graphics and replaces them with feature layers, primitive shape objects representing more complex in-game objects, while still maintaining some spacial aspects.

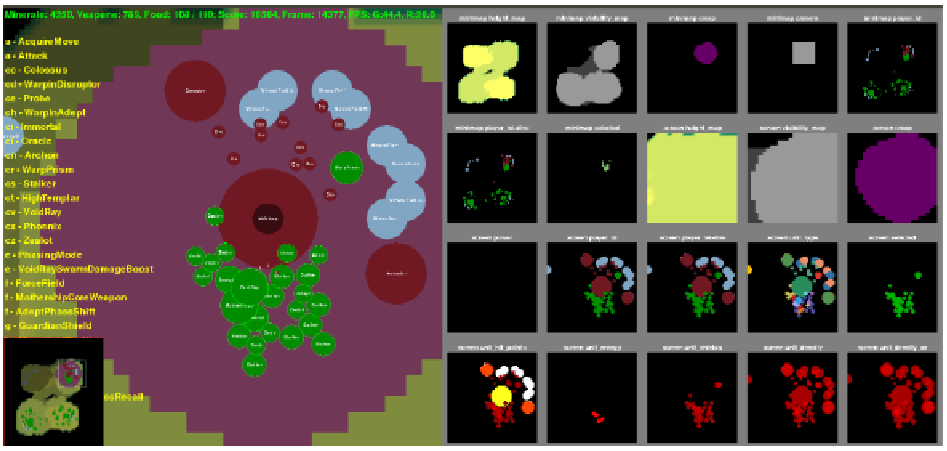

Figure 3.5: The feature layers of PySC2

In the figure above, on the left side, we can see a specific type of attack unit being represented as the green circles, resources as grey circles, worker units as small red circles etc. On the right side are different representations of features available from the full resolution mini-map, including height (top left), visibility (second from top left) and unit hit points or lives (bottom right).

SC2LE provides a set of 'mini-games,' which are stripped down versions of the original game intended to be much simpler scenarios, focused on obtaining a more fine grained objective. This allows training agents in progressive steps, building up to the ability to play a full game against multiple players.

A set of benchmark results were published by (Vinyals et al. (2017)) with the release of SC2LE. These results were obtained by a DeepMind RL agent trained using the A3C algorithm under 3 different network architectures. Although the results were underwhelming - the agent did not win a single game against the easiest built-in game AI, the fully convolutional network managed to utilize one of the unit's (Teran worker) abilities to move buildings out of attack range, thus managing a draw by surviving past the 30 minute time limit. However, no agent managed to devise a winning strategy. The ability to devise such an agent is still an open question.

3.4.3 Performance Metrics

The most popular, general performance metrics used in (Mnih et al. (2013)), (Van Hasselt et al. (2016)), (Wang et al. (2015)), (Mnih et al. (2015)) and (Mnih et al. (2016)) when evaluating model performance on the Atari 2600 test bed are listed below.

- Game score, the range of which will vary greatly between different games.
- Max Q-Value estimates
- Number of game frames survived

These metrics are gathered by running a predetermined number of evaluation games; where the agent selects it's action from the trained model. They are usually presented as a mean average, however there is an argument for taking the median value to lessen the effect of outliers (Bellemare et al. (2013)). The evaluation runs are held after a regular number of training updates, where the agent solely exploits the trained policy and no learning updates take place. These intervals vary between different papers.

Normalizing Scores

The scales for scoring between two games can vary greatly, which makes it difficult to compare the performance of an algorithm on different games by quoting score alone. (Bellemare et al. (2013)) recommend using normalizing all scores using predetermined reference values.

$$z_{g,i} = \frac{s_{g,i} - r_{g,min}}{r_{g,max} - r_{g,min}} \tag{7}$$

Where $z_{g,i}$ is the normalized score s_i in game g. $[r_{g,min}, r_{g,max}]$ are reference values that we normalize to. These could be the minimum and maximum obtainable scores in the game, or more interestingly (Mnih et al. (2015)) propose normalizing scores to the score achieved by a human player, where $r_{g,min} = \text{random players score} \text{and} r_{g,max} = \text{human players score}$. We can then see the algorithms percentage performance over human level.

Average reward (normalized or not) and average frames survived are an interesting duo of performance metrics. Together, they can give a better understanding of *what* the agent is learning to do. In some games, the score might be dependant on surviving for a long time, for others it might be to shoot as many 'things' as possible and rack up points. For this project, we use the normalization to human scores proposed by (Mnih et al. (2015)), average frames survived and model loss as a basis for our discussion. The human player in this case will be the author.

4 Design

5 Evaluation

6 Conclusion

Bibliography

Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017. URL `http://arxiv.org/abs/1708.05866`.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html`.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

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

Claude E. Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950. URL `https://doi.org/10.1080/14786445008521796`.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3): 279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL `https://doi.org/10.1007/BF00992698`.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

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

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