Carlos Purves

The PlayStation Reinforcement Learning Environment

Computer Science Tripos – Part II

Jesus College

Proforma

Candidate Number:	**2384E**
College:	**Jesus College**
Project Title:	**The PlayStation Reinforcement Learning Environment**
Examination:	**Computer Science Tripos – Part II, July 2019**
Word Count:	**11787**[1]
Line Count:	**4933**[2]
Project Originator:	Dr Petar Veličković
Supervisors:	Cătălina Cangea & Dr Petar Veličković

Original Aims of the Project

The project aimed to build a reinforcement learning environment for PlayStation games by modifying an emulator and implementing inter-process communication between this and a Python library. I planned to interface this with the OpenAI Gym API. As a method for providing initial results in this environment, Deep Q-Learning would be implemented to train agents using a variety of state representations.

Work Completed

The project was a substantial success. I have made modifications to an existing PlayStation emulator, produced a PlayStation Python library, built an interface to this library that conforms to the OpenAI Gym specification and trained several reinforcement learning agents to play it. Through extensive experimentation, I evaluated both the performance of the environment and the proficiency of the agents that I built.

Special Difficulties

None.

[1]Computed using `TeXcount` including footnotes and tables.
[2]Computed using `cloc` and `diff`. Around 80% Python and 20% C.

Declaration

I, Carlos Purves of Jesus College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Carlos Purves of Jesus College, am content for my dissertation to be made available to the students and staff of the University.

Date 17 May 2019

Contents

Acknowledgements

I would like to thank Cătălina Cangea and Petar Veličković for providing excellent advice and guidance during my project and being very helpful in reviewing this dissertation.

Chapter 1

Introduction

This dissertation describes the process of designing and implementing an entirely new platform for reinforcement learning research. The outcome of my work is a reinforcement learning environment for PlayStation (PSX) games with full audio, visual and byte-addressable memory support. I evaluate this environment by implementing several well-known reinforcement learning approaches on the PSX game "Kula World" and, for the first time, I implement an approach to reinforcement learning that incorporates sound as a form of state.

Reinforcement Learning (RL) describes a form of machine learning in which an agent learns how to interact with an environment through the acquisition of 'rewards' that encourage good behaviour and discourage bad behaviour. Over recent years, this model of learning has been shown to be effective in many real-world areas, including in self-driving cars [21], traffic control [1], advertising [10] and robotics [11]. While these applications are becoming increasingly prominent, most of the use-cases for RL are currently in gaming. Computer games represent an ideal test environment for RL, exhibiting clear notions of reward and often having a clearly defined end state.

Properly evaluating RL algorithms has often been a challenge. Initially, they were tested within purpose-built environments[1]. However, this approach has been considered flawed:

> Ideally, the algorithm should be compared across domains that are (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.
> — *Bellemare et al. [3]*

In order to improve the process of RL evaluation, researchers designed an environment known as *Arcade Learning Environment* ALE [3] in 2012. ALE is a modification of an Atari-2600 emulator known as *Stella*, providing a software interface to control and read data from Atari-2600 games. With ALE, researchers can train agents on any of the 500 [12] games available for the Atari-2600 console, using the console's RAM contents and visual output to inform training.

In 2015, DeepMind [15] made use of ALE to perform Deep-Q Learning, a technique that involves the use of neural networks to interpret state and choose actions. The results were very promising, with the agent *outperforming* human expert players in 22 out of the 49 games that they trained on. In 2016, OpenAI announced OpenAI Gym [4], which allows researchers and developers to interface with games and physical simulations in a standardised way through a Python library. Gym now represents the de-facto standard evaluation method for RL algorithms [2]. It includes, amongst others [16], several Atari-2600 games which utilise a modification of ALE [17].

Since OpenAI Gym has been available for use, RL algorithms have become much better. Advances such as Double DQN [8], Prioritised Experience Relay [19] and Duelling Architec-

[1]An example of a library from this era is `http://mmlf.sourceforge.net` from 2011.

tures [22] have brought improvements over DQN on Atari games[CITE]. Policy-based methods such as A3C have brought further improvements [14] as well as asynchronous methods to RL.

Despite all of the iterative improvements in reinforcement learning, many evaluations still centre around games built for the Atari-2600 console. While these games are often very good candidates for RL, many do not provide it with a significant challenge: in 2015, DeepMind's DQN agent could already play 22 out of the 49 ALE games better than human expert testers [15].

The Atari-2600 console has only 128 bytes of RAM, 128 displayable colours and a 1.19MHz CPU. The Sony PlayStation, however, is substantially more powerful, with 2 megabytes of RAM, 16.7 million displayable colours and a 33.9 MHz CPU. The range of titles available for the PlayStation is also much greater, totalling almost 8000 worldwide [9] compared to 500 for the Atari-2600.

This project increases the complexity and quantity of games that can be used for RL by designing the first ever RL environment in which arbitrary PlayStation games can be controlled and observed through a Python interface. Such an interface provides new opportunities for research into RL, including the use of multi-modal state spaces that incorporate audio and dual-shock (vibration) feedback. The PlayStation also poses new challenges for RL: enabling environments that are both more difficult and have more complex representations. For the first time in RL, I train and evaluate agents on a PlayStation game using *Q-Learning*.

Chapter 2

Preparation

In this chapter, I outline the theory surrounding reinforcement learning and introduce a formalised model for games. I then discuss related work and the way in which the formalisation applies to more complex games. In §2.4, I introduce Kula World—a PlayStation game—and describe an abstraction of it that conforms to the formalisation. §2.5 describes how an implementation of the abstraction may be realised using a PlayStation emulator. I then present a requirements analysis and conclude with a brief discussion of tools and practices.

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is defined by:

- $\mathcal{S}$: The set of states in which the system can exist.
- $\mathcal{A}$: The set of possible actions that can be taken by an agent with control of the system.
- $\mathcal{T}(s, s'|a) = \mathbb{P}(s_{t+1} = s'|a_t = a, s_t = s)$: The transition probability between states s and s' as the result of the agent taking action a.
- $\mathcal{R}(s, s') = \mathbb{E}(r_t|a_t = a, s_t = s)$: The expected reward gained by the agent controlling the system as a result of the system transitioning from state s to s'

Also, MDPs exhibit the *Markov Property*:

$$\mathbb{P}(s_{t+1} = s|\underbrace{[s_t, s_{t-1}, s_{t-2}, \ldots]}_{\text{previous states}}, \underbrace{[a_t, a_{t-1}, a_{t-2}, \ldots]}_{\text{previous actions}}) = \mathbb{P}(s_{t+1} = s|s_t, a_t) = \mathcal{T}(s_t, s_{t+1}|a_t). \quad (2.1)$$

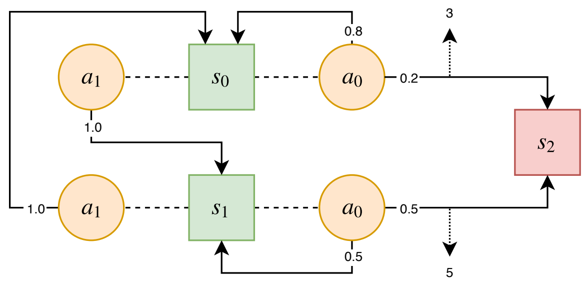
An MDP may include special states, such as an initial state s_{initial} and a set of terminating states $\mathcal{S}_{\text{term}}$. When both exist, a set of transitions from a system's initial state to one of its terminating states constitutes an *episode*. An example MDP is shown in Figure 2.1a and the view of interactions with the system of the agent is shown in Figure 2.1b.

2.2 Reinforcement Learning

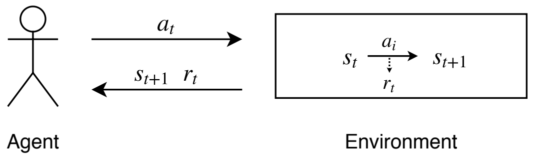
Reinforcement Learning describes a process in which an agent uses interactions with an MDP to learn a *policy* $\pi : \mathcal{S} \to \mathcal{A}$ that maps a state to an action. Given some policy π, an agent should take an action $\pi(s)$ when in state s. A policy is *optimal* if, for all episodes of length n, it maximises the expected cumulative reward:

$$R_\pi = \mathbb{E}_\pi \left\{ \sum_{t=0}^{n} \mathcal{R}(s_t, s_{t+1}) \right\}. \quad (2.2)$$

(Where s_{n+1} is a terminating state and $s_1, s_2, \ldots, s_n$ are all chosen according to π.)



(a) An MDP.



(b) An agent interacting with an MDP.

Figure 2.1: Figure 2.1a shows an MDP with three states—of which s_2 is a terminating state—and two actions. The action a_1 acts as a toggle between states s_0 and s_1 (and can be taken while in either of them). Taking a_0 in either state causes a transition to s_2 with a certain probability. Rewards are represented by arrows pointing from state transitions. Figure 2.1b shows the canonical form of an MDP, in which an agent performs an action a_t on an environment causing it to undergo an internal state transition $s_t \to s_{t+1}$. This transition yields a reward of r_t.

Many methods exist to learn policies; however, the methods that this project uses are entirely *value-based*. For these, π is learned by estimating a *Q-value* $Q(s|a)$ for each state s and action a pair, defined by:

$$Q(s|a) = \mathbb{E}_\pi \left\{ \sum_{t=0}^{n} \gamma^t \mathcal{R}(s_t, s_{t+1}) \;\middle|\; a_0 = a, s_0 = s \right\}. \tag{2.3}$$

(Where $0 \le \gamma < 1$ is known as the *discount factor*.)

The value of γ is chosen to control how short-sighted the agent is. Values that are low represent a policy that cares only about immediate rewards, whereas values that are high represent policies that are far-sighted[1].

Given a policy π, actions are chosen such that:

$$\pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q(s|a). \tag{2.4}$$

Suppose we have some deterministic MDP having

$$\mathcal{G}(s, a) = s' \Leftrightarrow \mathcal{T}(s, s'|a) = 1, \tag{2.5}$$

and using Property 2.1[2], we can rewrite Definition 2.3 as:

$$\begin{aligned} Q(s|a) &= \mathbb{E}_\pi \{ \;\; \mathcal{R}(s_0, s_1) + \gamma Q(s_1|a_1) \;\;|\;\; a_0 = a, s_0 = s \;\; \} \\ &= \mathcal{R}(s, \mathcal{G}(s,a)) + \gamma Q\left(\mathcal{G}(s,a) | \pi(\mathcal{G}(s,a))\right) \\ &= \mathcal{R}(s, \mathcal{G}(s,a)) + \gamma Q\left(\mathcal{G}(s,a) | \operatorname*{argmax}_{a' \in \mathcal{A}} Q(\mathcal{G}(s,a)|a')\right) \\ &= \mathcal{R}(s, \mathcal{G}(s,a)) + \gamma \max_{a' \in \mathcal{A}} Q(\mathcal{G}(s,a)|a'). \end{aligned} \tag{2.6}$$

[1]Values of $\gamma \approx 1$ can cause instability when an episode's length is large, as changes in the value of Q anywhere in the state-space have a significant effect on all others.

[2]This allows us to separate policy decisions taken at a time $t+1$ from those taken at time t, providing that the starting state of the former is the ending state of the latter.

From this, we can deduce the value of $Q(s, a)$ using an iterative approach: when an MDP in state s transitions to s' as a result of action a with reward r, update the estimate of Q using the relation:

$$Q_{i+1}(s|a) = (1 - \alpha)Q_i(s|a) + \alpha \left\{ r + \gamma \max_{a' \in \mathcal{A}} Q(s'|a') \right\}. \tag{2.7}$$

(Where α specifies a *learning rate* to account for stochastic systems.)

2.3 Games as Markov Decision Processes

In order to discuss RL within the context of games, it is important to formalise the notion of a game. In the context of gameplay, the agent may be referred to as a *player*. For the purposes of this project, a game is defined by:

- A: The set of actions that a player can take
- B: The set of states that a game can exist in
- $G(s, a)$: The *game function* describing the state that a game in state s transitions to after a player performs action a.
- $R(s, a)$: The *reward function* describing the reward gained by a player after a player performs action a on a game in state s.

An agent playing a game of this form is shown in Figure 2.2. This definition describes a game as a subset of deterministic MDPs. Hence, we can rewrite Equation 2.6 as:

$$Q(s|a) = R(s, a) + \gamma \max_{a' \in A} Q(G(s, a)|a') \tag{2.8}$$

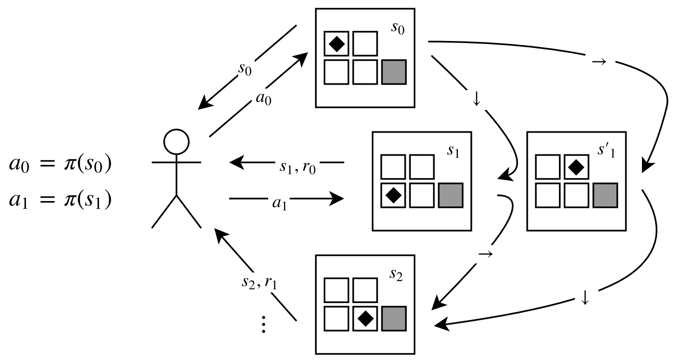


Figure 2.2: An agent playing a game. The agent uses its policy π to determine which action to take at each state. Shown are two possible sets of actions that transition the game from state s_0 to s_2. This is encoded by the game function in the equality $G(G(s_0, \downarrow), \rightarrow) = G(G(s_0, \rightarrow), \downarrow)$. Note that because of Property 2.1, both the game function $G(s, a)$ and the policy π will be the same when applied to the state after either set of actions have taken place.

While this formalisation is trivially applicable to simple games, others may require some translation in order to conform to it. Specifically, games that display multiple frames per second often suffer from two main problems: the inability for actions to be carried out at every single frame change; and the inability‘ of a single frame to represent what is happening in the game. To counter the first problem, [15] employed a *frame skip* of 4, which sets the environment to discard all but every fourth frame. To counter the second, [15] used *frame stacking* in which every group of four frames would be combined together (or *stacked*) to describe the state.

2.4 Kula World

The game that I used for this project was *Kula World*, a game developed by Game Design Sweden A.B. and released in 1998. It principally involves controlling a *ball* (or *player*) within a *world* (or *level*) containing a series of *objects*. The world is made up of a *platform* on which the ball can move. These features are shown in Figure 2.3. An object can be a coin, a piece of

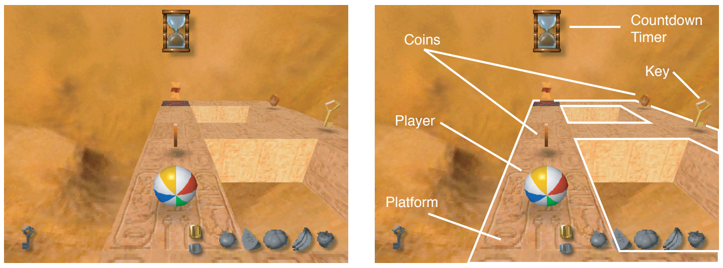


Figure 2.3: Level 1 of Kula World (left) and annotated (right).

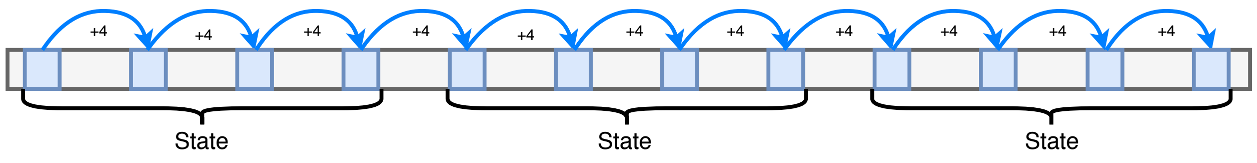
fruit or a key, each of which a player can collect by moving into them. The ball is controlled through the use of the *directional pad* (D-Pad) and the *cross* button. Pressing the right or left directional button rotates the *direction of view* (or *direction*) 90°clockwise or anti-clockwise in the line perpendicular to the platform, respectively. Pressing the up directional button moves the player forwards on the platform, in the direction that the camera is facing. Pressing the cross button makes the ball *jump*. This can be pressed simultaneously with the forward directional button to *jump forwards*. *Jumping forwards* moves the player two squares forwards, over the square in front of it. If the platform does not exist at the destination of a jump, that jump will result in the player losing the game.

The definition of the action space A for Kula World is relatively simple. I omitted including a *jump* action since this served no purpose to the game on the levels that I tested. In total, the action space is given by:

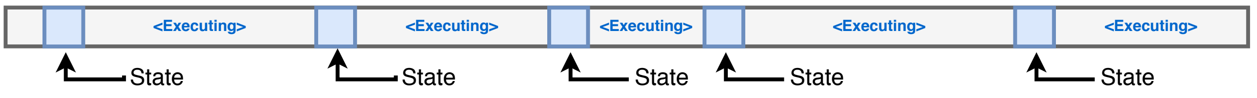
$$A = \{\textsc{Forward},\quad \textsc{LookRight},\quad \textsc{LookLeft},\quad \textsc{JumpForward}\} \tag{2.9}$$

There is a clock on each level, which counts down in seconds from a start value which depends on the level. To complete a level, the player must pick up all of the *keys* and move to a dedicated goal square before the clock reaches zero. Collecting objects gains points for player which are added to the *score* for the level. Collectively, these make up *game variables*. The most important such variables are `score`, describing the number of points a player has gained; `time`, the amount of time left on the count-down clock; and `status`, whether the game is still ongoing or the player has lost. I later used these variables to construct the reward function R.

Kula World, as with almost all PSX games, supports running at up to 30 frames per second. However, it is not suitable to employ *frame skip* in this situation, since moves have variable lengths. A jump, for example, takes roughly a second longer than rotating the camera. Lengths of moves can depend on the specific state of a level; for example, moving forward to collect a coin takes longer than moving forward without collecting a coin. This is a problem since a move cannot be carried out while another is taking place. Further, since the logic for the game takes place within the CPU of the console, it is not possible in general to predict the duration



(a) A visualisation of frame skip and frame stacking. In [15], only every fourth frame is considered and of those, every four frames are combined (stacked) into a single state representation. New frames are requested by the Python library upon each action, making this approach synchronous.



(b) An asynchronous approach to frame skipping. In environments where actions are long or have variable length, the state transition occurs asynchronously. The transition ends once the immediate effects of the associated action have ceased.

Figure 2.4: A comparison of the state-delimiting techniques used in [15] with those used in this project. Note that frame stacking is not used in the latter case.

of a move prior to it completing. As a result, a frame skip approach would require skipping enough frames so that *any possible* action could have been completed.

In order to satisfy the requirements of the game formalisation from earlier, the game function G for Kula World needed to account for an arbitrary amount of time passing between each consecutive state, during which the console will be executing asynchronously. A comparison between both frame skipping (synchronous) and the asynchronous approach is shown in Figure 2.4. In this sense, Kula World represents a far more complex setting than games on which RL is usually carried out.

I chose to represent state in three different ways:

- The *Simple* representation involves each state being uniquely defined as an integer. Each state is entirely independent of all others and there is no relationship between spatially adjacent states.
- The *Agent-Local* representation is a cropped and rotated map of the level, relative to the position and direction of the player.
- *Complex* states can include visual or audio data.

2.5 Emulation

In order to perform RL on PSX games, I needed to be able to emulate a PSX console concurrently with the learning algorithm. I chose to follow a similar approach to that of ALE, by taking an existing emulator and modifying it. The primary requirements for this modified emulator were that it should run on Linux, support programmatic input, support exporting PSX frame buffer contents and run in a headless mode. It also needed to be entirely self-contained in a folder, both for portability and to avoid needing `sudo` privileges for installation.

After spending time investigating possible emulators, I decided to use PCSXR[3], an open-source emulator created in 2009. PCSXR is a fork of an earlier project: PCSX, which was originally

[3]Available at `https://github.com/pcsxr/PCSX-Reloaded/`

built in 2000. Much of the original code from PCSX still exists in PCSXR with heavy modification. The advantage of this is two-fold: it is highly compatible with most PSX games and it has a modular design, with many different plugins being used for different elements of the emulator. However, there is no documentation whatsoever, with much of the code having been written without comments and being stylistically inconsistent. This often meant that I had to work backwards through the code to find the locations of important features, taking up a significant amount of time in the early stages of the project.

2.6 Requirements Analysis

This project aimed to develop a Python library that can be used to control PSX games and create an RL environment that conforms to the OpenAI Gym specification. These would be tested by implementing RL methods.

Considering all of the components required for this project to be a success, it was clear that it would represent a significant software engineering effort. In order to manage this, I ranked and analysed each component to produce Table 2.1.

	Component	Priority	Risk	Difficulty
1	Basic control of emulator from Python	**High**	**Medium**	**Medium**
2	Simulate a PlayStation controller	**High**	**High**	**Medium**
3	Locate relevant game variables in memory	**High**	**Very High**	**High**
4	Obtain screenshots from the emulator	**High**	**Medium**	**Medium**
5	Handle asynchronous state transitions	**High**	**High**	**Medium**
6	Support freezing/unfreezing the emulator	**Medium**	**High**	**Medium**
7	Support for concurrent emulators	**Low**	**Low**	**Medium**
8	Obtain and process audio from emulator	**Low**	**High**	**High**
9	Simple Q-Learning agent	**High**	**Low**	**Low**
10	Agent-Local DQN agent	**High**	**Low**	**Medium**
11	Complex (visual) DQN agent	**High**	**Medium**	**High**

Table 2.1: Requirements analysis table.

2.6.1 Contingency Planning

Several possible contingency plans were made for the above criteria. For example, failing to locate the score variable in memory could have been mitigated through the use of Optical Character Recognition (OCR) on the display output. Also, several approaches to exploit the X windowing environment to simulate keystrokes and obtain screenshots were investigated, in case direct approaches were not possible. The ability to freeze and unfreeze the execution of the emulator could, with some trade-offs[4], be replaced with toggling the in-game pause menu. None of the contingency plans needed to be used.

[4]These trade-offs come from the fact that in-game pause menus do not freeze CPU, RAM or frame-buffer state, a possible cause of inconsistency between states.

2.7 Choice of Tools

Some of the important libraries and applications required for this project are listed below:

- **C**: The emulator is written entirely in C.
  - `OpenGL`: The PSX GPU is rendered using OpenGL, so modifications were required to support windowless execution.
  - `SDL`: Many of the key components of the emulator utilise SDL, so modifying them required me to understand how it works.
  - `XVFB`: **X V**irtual **F**rame **B**uffer was a requirement in order to support the creation of X windows on remote machines.
- **Python**: OpenAI Gym is built in Python, making it my target language. Also, most machine learning libraries have Python interfaces.
  - `PyTorch`: for building neural networks, as it is very well documented and more simple to use than `TensorFlow`, a popular alternative.
  - `Matplotlib`: to visualise data and results. Specifically, I used it to visualise audio waveforms.
  - `NumPy`: to handle and manipulate data.
  - `SciPy`: for calculating statistical values from results.
  - `pandas`: to load and manipulate data that I had collected from experiments.
  - `sysv_ipc`[5]: which provided a Python interface to many of the System-V Inter-process communication features in Linux.
  - `python_speech_features`[6]: for audio processing during the later stages of the project.
  - `OpenCV`: to perform image pre-processing for the DQN agent as well as analysis throughout the project.
  - `Python Image Library`: to add support for storing images to disk in a variety of formats to the PSX Python library.

2.8 Starting Point

At the start of this project, I had very little experience in writing C and had never used neural networks in anything beyond a theoretical capacity. I had also never implemented any RL algorithms, nor had I built or modified an emulator. As a result, this project started with a long period of preparatory reading, in which I made extensive use of online sources. Overall, around 2 months at the start of the project was spent performing background reading. The starting point for the code in my project is the source code of PCSXR [7]. Everything else was written by me, including the implementations of neural networks and learning agents.

The project was made up of two main parts: the modified emulator and the Python PSX library. Since these work together and the performance of each is heavily dependant on the other, it was helpful to build both parts concurrently. I adopted an iterative approach to development in which I maintained full integration between the emulator and the library while adding features to both.

[5]Available at `http://semanchuk.com/philip/sysv_ipc/`

[6]Available at `https://github.com/jameslyons/python_speech_features/`

[7]Available at `https://github.com/pcsxr/PCSX-Reloaded/`

Chapter 3

Implementation

In this chapter, I discuss the implementation of my work. I first explain the structure of my project, then I discuss several large components sequentially, each with increasing abstraction from the console itself. I start with the lowest level console modifications and finish with the implementation of the learning algorithms and neural network architectures. Each section assumes the complete functioning of the sections before it.

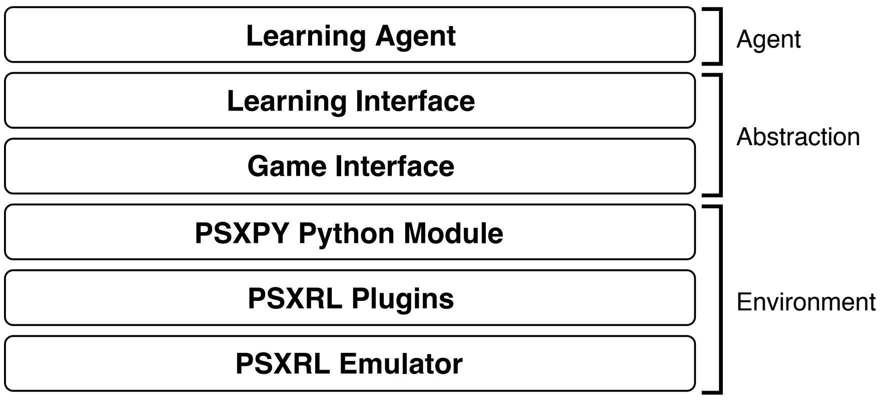


Figure 3.1: A high-level view of the component layers of my project. Layers that are categorised under *Environment* are built with the aim of developing a Python interface for the PSX console. Layers categorised as *Abstraction* are built with the aim of producing a Python interface to Kula World. The top layer refers to the implementation of the reinforcement learning agent.

The implementation of my project involved three main parts. Firstly, I was required to build a PSX environment which exposes a simple interface to the PlayStation console in Python. Secondly, I had to build an abstraction which, given some game and level, exposes a simple action/response interface through which to control gameplay. Thirdly, I was required to build a learning agent framework which utilises this interface for reinforcement learning purposes. This can be visualised as the structure in Figure 3.1.

I named the modified emulator `PSXRL` to reference its purpose: facilitating RL on the PSX console. The library which provides a Python interface to `PSXRL` was given the package name `PSXPY`.

3.1 Environment

In this section on the Environment, I begin by introducing the methods of communication between the emulator and the Python library. Then, I describe the protocols used over those communication channels. Finally, I describe the implementation of several important features that were added to the emulator in order to satisfy my requirements.

3.1.1 Communication

To build the environment, it was clear that a method of *Inter-Process Communication* (IPC) would be required since PCSXR is written in C whereas OpenAI Gym Environments are implemented in Python. My main requirements for the IPC channel were that it supports:

1. sending instructions to the virtual PSX controller component of PSXRL—which I call `FakeJoy`—including support for arbitrary combinations of button presses;
2. running concurrently alongside channels serving other instances of the environment;
3. sending control messages to the running emulator, such as to close it or free memory that it is holding;
4. callbacks so that synchronous (blocking) methods can be implemented within PSXPY;
5. a system that notifies the Python library of changes to the contents of arbitrary sections of memory within the console;
6. sending screenshots and audio data from the emulator to the Python library.

Method	Form
Pipe	Unidirectional communication for processes with common ancestry, reader and writer start together.
FIFO	Unidirectional communication between any processes, support for multiple writers, named, reader and writer(s) can be started independently.
Shared Memory	Bidirectional (non-thread-safe) communication between any processes, identified by a unique key, reader and writer(s) can be started independently, very fast throughput.
Temporary Files	Files stored (usually) in the `/tmp` directory on Linux machines. Can be deleted on reboot or when the file is closed. On some systems, temporary files may bypass being written to disk.

Table 3.1: A summary of some IPC techniques in Linux.

There are a number of techniques in Linux for IPC. These are summarised in Table 3.1. I used shared memory to transfer larger data due to its speed and ability to scale. I used a pair of FIFO pipes—`ProcPipe` and `CBPipe`—to facilitate bidirectional communication between PSXRL and PSXPY and an additional pipe for communication between PSXPY and the controller emulator — `FakeJoyPipe`. The latter was added to reduce congestion on the main bidirectional communication channel as well as to mitigate issues associated with concurrency. Concurrency issues arise since the controller is polled in a separate thread to the execution of the main emulator, so sharing a pipe would require additional concurrency handling within PSXRL to deliver messages to the correct thread. The layout of IPC that PSXRL and PSXPY use is shown in Figure 3.2.

3.1.2 Protocol Design

The communication protocols were designed to achieve both low latency and fairness. Latency was considered so that the emulator could respond to actions by the agent quickly, without

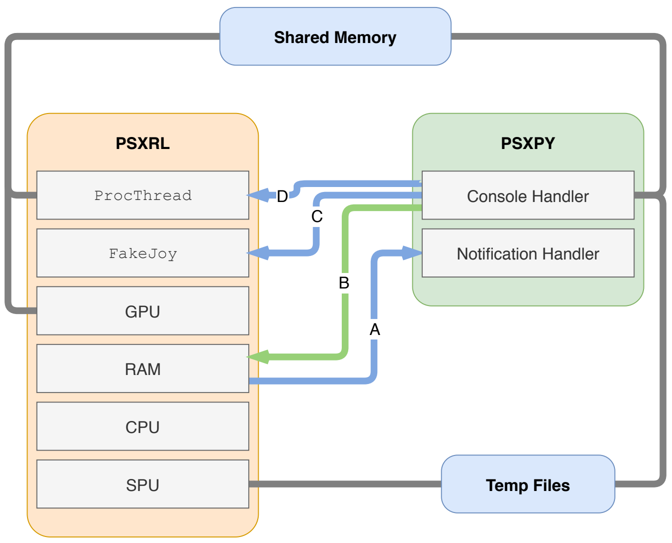


Figure 3.2: The IPC layout for PSXRL and PSXPY. Pipes are coloured green, FIFO queues are coloured blue and Unix `fopen` calls are coloured grey. **A** is `CBPipe`, **B** is the Unix standard input (`stdin`) pipe, **C** is `FakeJoyPipe` and **D** is `ProcPipe`.

wasting time within the game; and fairness is important because there will be many different operations being executed simultaneously, of which some may send messages very frequently. For example, updates to score are rare, whereas updates to camera position are very frequent.

`FakeJoyPipe`: Communication from PSXPY to `FakeJoy`

`FakeJoyPipe` carries individual button action instructions, in order to satisfy Requirement 1. I minimised the size of packets to a single byte, reducing the latency that would be incurred in reading and decoding each instruction. There are 17 physical buttons on the PSX controller, so a button can be uniquely identified by just 5 bits. In order to support variable contact length button presses, I split the instructions into `depress` and `release` operations. To add support for the two controllers of the PSX, there is an additional *controller discriminator* bit. The resulting instructional schema is shown in Figure 3.3.

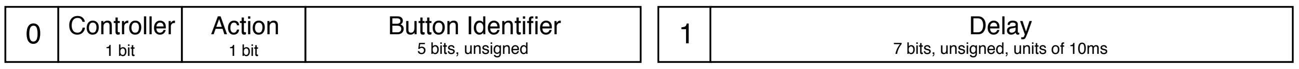


Figure 3.3: Specification for `FakeJoyPipe`, giving forms for both issuing controller instructions and for imposing fixed time separations between such instructions.

`ProcPipe`: Communication from PSXPY to PSXRL

It was important that `ProcPipe` used a variable schema, because there are sixteen different instructions that must be accounted for and because smaller packet sizes help to reduce latency. A full specification for `ProcPipe` is given in Appendix B. `ProcPipe` satisfies Requirement 3 from above and provides the method by which Requirement 1 in §2.6 is satisfied.

`CBPipe`: Communication from PSXRL to PSXPY

The primary requirement of `CBPipe` was that it be able to carry information about memory changes (*memory notifications*) from the PSX memory emulator to PSXPY, satisfying Requirement 5. However, I realised that I could use the same pipe to transfer more general notifications to PSXPY using the same pipe—satisfying Requirement 4—with only very minor concurrency control through the use of a single `pthread_mutex`. A full specification for packets sent over `CBPipe` is given in Appendix B.

`stdin`: Initialisation of PSXRL from PSXPY

Much of the initialisation of PSXRL can be carried out through the use of command-line arguments, however, *memory listeners* are an example of a feature that requires a more structured approach. Memory listeners watch a certain section of memory for changes and send a notification back to PSXRL when they occur. A memory listener can be defined by:

- k: a unique value by which the listener can be identified in notifications.
- i: the start (in bytes) index of the section of memory that the listener should watch.
- w: the size (in bytes) of the section of memory that the listener should watch.
- `flow_limit`: a Boolean value indicating whether the listener should implement flow control when sending notifications.
- `fresh_only`: a Boolean value indicating whether the listener should send notifications about memory locations that have *been modified* but *not changed in value.*
- `asleep`: a Boolean value indicating whether the listener should suppress all notifications when it sees changes. This is the only mutable property of a listener and can be changed at any time during execution.

When *flow limiter* is set to *on*, listeners will send notifications at a maximum rate of 2 per 0.4-second interval. This is an attempt to maintain fairness between different notification types.

The definition of a listener is encoded in a packet called a *Memory Listener Declaration.* The specification for such packets is given in Figure 3.4.

Start Index	Length	Key	Fresh Only	Flow Contr	Zero
(4 Bytes; HBO Integer)	(4 Bytes; HBO Integer)	(1 Byte)	(1 Byte)	(1 Byte)	(1 Byte)

Figure 3.4: The specification used for memory listeners sent through `stdin`.

Concurrency Support

To support multiple concurrently training agents, each instance of PSXRL must have a unique set of FIFO pipes through which to communicate with their respective PSXPY sessions. To implement this, PSXPY passes the prefix for FIFO names as a command line argument to PSXRL. Pipes are differentiated by adding "-proc", "-joy" and "-mem" for `ProcPipe`, `FakeJoyPipe` and `CBPipe` respectively. This satisfies Requirement 7 in §2.6.

3.1.3 Memory Listeners

I initially investigated polling as a method to implement memory listeners; this involved PSXPY requesting—through `ProcPipe`—that the console's memory layout be copied into shared memory. PSXPY would then compare that copy of memory to the copy that is had saved from the previous poll. This solution was very slow, as it required repeatedly comparing two arrays of 2 million bytes. Instead, I chose to integrate memory listeners with the emulator's memory write functions.

In addition to being sent over `stdin`, listeners must be specified in PSXPY *before* PSXRL is run. This allows PSXRL to use a contiguous space in memory to store them, which means that traversing through all listeners doesn't incur penalties due to pointer indirection. This is important since memory write instructions occur both synchronously and very frequently so additional complexity here could lead to a slow-down in the emulator's overall execution. Listeners can be sent to sleep or woken at any time, using an instruction sent over `ProcPipe`.

To implement the flow control features that are present in the Memory Listener Declaration protocol, I associated a finite state with each listener in PSXRL.

- `SILENCED` indicates that a listener is *asleep*. Notifications will not be sent from listeners that are asleep.
- `PUSHED` means that a notification has been sent within the last 0.4s time interval recently for the listener.
- `UNPUSHED` means that there are changes in memory that have been detected, but their notifications are queued.
- `NO_CHANGES` indicates that, if any change that is detected, the listener should send a notification.

Each time a memory write occurs, the memory controller performs the routine shown in Figure 3.5. The presence of flow control means that two different threads can prompt notifications being sent: the main thread, where notifications are sent in response to changes in memory; or within the flow control thread, which resets listener notification queues every 0.4 seconds. Each of these scenarios will cause the state of a listener to change, so a lock (using `pthread_mutex`) must be associated with each listener to prevent it having its state written to simultaneously by both threads. Each listener also has an associated cache value, which holds a copy of the contents of the memory location that a listener is attached to. This field is only updated if `fresh_only` is set and is used to detect whether the value at a location in memory has been changed by a write.

3.1.4 Synchronous Methods

There are certain methods within PSXPY that should occur *synchronously* with actions in PSXRL. These include `pause`, so that PSXPY can safely read memory data with a guarantee that it will not change; `resume`, to ensure that button presses will definitely be responded to; and `loadState`, so that actions are only be carried out once a state has finished loading.

In order to implement notification callbacks, PSXPY maintains a table that maps each *notification identifier* to the time at which the last notification with such an identifier was received. To enforce blocking until the receipt of a certain notification, it is sufficient to wait until the corresponding time value increases. This is shown in Algorithm 1.

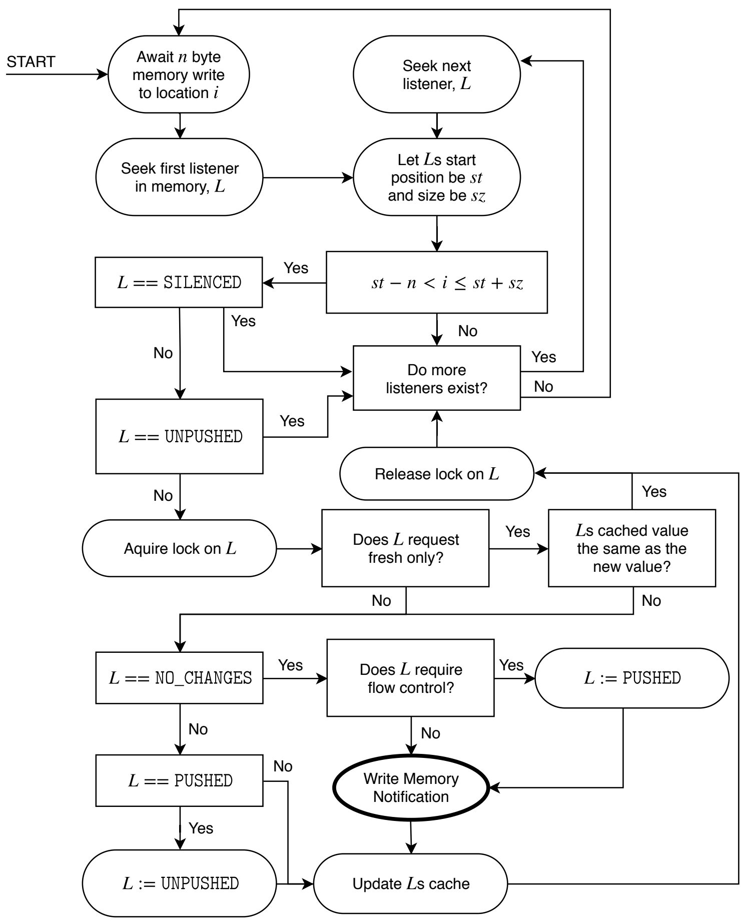

START
Await n byte memory write to location i
Seek first listener in memory, L
Seek next listener, L
Let Ls start position be st and size be sz
$st - n < i \leq st + sz$
Yes
No
L == SILENCED
Yes
No
Do more listeners exist?
Yes
No
L == UNPUSHED
Yes
No
Release lock on L
Aquire lock on L
Does L request fresh only?
Yes
No
Ls cached value the same as the new value?
Yes
No
L == NO_CHANGES
Yes
No
Does L require flow control?
Yes
No
L := PUSHED
L == PUSHED
Yes
No
Write Memory Notification
L := UNPUSHED
Update Ls cache

Figure 3.5: Memory Listener Flowchart. The process starts on seeing a memory write, after which it iterates through each listener and checks whether the write occurred in any listener's specified section of memory. If so, it chooses whether to send a notification based on that listener's properties.

3.1.5 Emulator Additions

In order to add support for the features in the specifications above, I had to perform several modifications to the emulator. This was a difficult task due to the lack of any documentation for PCSXR; however, I attempted to retain the existing structure and style whenever possible.

`FakeJoy`: A Simulated PSX Controller

In order to emulate PSX controller button presses, I modified the existing SDL controller interface. In addition to polling for inputs from the keyboard, `FakeJoy` reads from `FakeJoyPipe`. When it receives an instruction, it parses it to recover the button identifier, controller and action required, then calls the required methods as it would with keyboard input, before awaiting more instructions over `FakeJoyPipe`. The implementation of `FakeJoy` functions in satisfaction of Requirement 2 in §2.6.

Algorithm 1: Approximating Synchrony Using Asynchronous Notifications

```
Initialisation:
for $i \in \mathbb{N}$ do
    last_updated[i] = 0;
end
Methods:
def receiveNotification(k: key):
    last_updated[k] ← now();
def await(k: key, timeout: int):
    limit ← now() + timeout;
    start ← last_updated[k];
    while $last\_updated[k] \leq start$ and $now() < limit$ do
        continue;
    end
    if $cached = last\_updated[k]$ then
        return TIMEOUT;
    else
        return SUCCESS;
```

Figure 3.6: The implementation of synchronous functions within PSXPY. An instruction is first passed from PSXPY to PSXRL over `ProcPipe`, then PSXPY calls `await` on the notification identifier that represents the completion of the instruction. This call will block until the notification is received and the instruction is complete. `now` represents the integer number of seconds that have elapsed from some fixed time.

State Saving and Loading

In order to be able to both quickly restart levels after they finish, as well as to support training on arbitrary levels, it was important that the abstraction could quickly move the game to a pre-determined state. To support this, I modified an existing function in PCSXR that allowed users to save and load states made up of register values and memory contents to and from disk. The modification required moving the calls to these functions into `ProcPipeThread` and resolving associated race conditions, as well as building custom save and restore functions which could be executed through `ProcPipe`.

Changing the Speed of the Console

In general, RL requires a large number of episodes to converge on a policy. As a result, training would greatly benefit from the console running at a faster-than-real-time speed. In addition, there is a trade-off between running the emulator quickly and using up as little time as possible *within the game*[1]. I was able to use the frame-rate multiplier within PCSXR to build the `setSpeed` method of PSXPY. This allows users to specify the execution rate of the CPU as a percentage of its default.

[1]This a trade-off discussed more carefully in §4.2.5.

The console can also be *frozen* during execution through the use of a blocking mechanism within the main thread of PSXRL. To freeze the emulator, a flag is set by `ProcPipeThread`. On each CPU cycle, if this flag is set, the main thread will repeatedly pause for 0.1 seconds in between checking whether the flag is set. This satisfies Requirement 6 in §2.6.

Capturing Screenshots

Since the emulator renders graphics using OpenGL, I was able to utilise `glReadPixels` to copy pixel data into shared memory. To prevent tearing, in which the screen would be captured between frame updates, `glReadPixels` is only called once a frame has been fully rendered. This is implemented through the use of a flag, `takeScreenshot`, which is set to `True` when a screenshot is requested through `ProcPipe` and to `False` once the frame has been rendered and the pixels copied to shared memory. The outcome of the implementation satisfies Requirement 4 in §2.6.

Reading Memory

In addition to memory notifications, which return the memory contents of locations that have listeners attached when writes take place there, it is important that PSXPY support reading arbitrary sections of memory on demand. Reading sections from memory is achieved by sending an instruction, through `procPipe`, that specifies the start index and size of the section that is to be read. These values are then used with `psxMemPointer` (an existing method in PCSXR) to copy the specified section into shared memory, where it can be accessed by PSXPY.

Capturing Audio

Capturing audio represented my primary extension objective. To achieve this, I modified the Sound Processing Unit (SPU) of the emulator, via the SDL sound plugin. Recordings are controlled by a flag, `spuShouldRecordAudio`, which can be set by sending an instruction through `procPipe` along with a file name. When a recording is taking place, each byte of output that gets sent, using SDL, to the system's audio driver gets copied and appended to the temporary file. This can lead to large file sizes, so three features were added to mitigate large sizes:

- **Early Amplitude-Based Cropping**: audio only begins recording when a certain amplitude threshold is crossed in the output audio of the console.
- **Amplitude-Based Termination**: recording stops after a specified period of time during which the output is below a certain threshold.
- **Late Amplitude-Based Cropping**: recordings are cropped to within a specified period of time of the last measurement which had an amplitude greater than a certain threshold.

In my testing, not using these features would often result in large amounts of silence being recorded due to the latency involved in starting and stopping recordings over `ProcPipe`. The audio recording implementation satisfies Requirement 8 in §2.6.

3.2 Abstraction

This section describes the implementation of the abstraction layers shown in Figure 3.1. Firstly, I describe the methods in which game variables were located, pursuant to Requirement 3 in §2.6. Then, §3.2.2 describes the way in which notions of game levels are incorporated into the abstraction. I then go on to discuss the semantics of performing game actions within the abstraction in §3.2.3, including an implementation that satisfies Requirement 5 regarding transition delimitation in §2.6. In §3.2.5, I describe how state information is retrieved and interpreted by the abstraction. Finally, §3.2.8 describes how this abstraction is designed to conform to the OpenAI Gym specification.

3.2.1 Finding Game Variables

In order to properly implement Kula World, it was very important that I was able to locate and read game variables to use in allocating reward to the agent. The process of finding these took around 4 weeks and was performed early in the project to give me time to seek out alternatives (such as the use of OCR) if they were required. I attempted to locate documentation of the PSX's memory layout but was only able to find a very limited amount of information, none of which gave me any better indication of where an executable may store its variables. As a result, the search space was 2 million bytes in size.

Methodology

Here I discuss the process of locating the score variable, but the method is similar for each of the variables. The first approach I took was to modify the emulator's CPU component to output register values at regular periods. I then plotted the values of the program registers in an attempt to find registers that correlated with the score. This was unsuccessful, with register values proving to be too volatile for any useful analysis.

I then decided to utilise PSXRL's memory reading features. This allowed me to 'record' memory contents by outputting the entire memory contents to disk every 0.1 seconds, often resulting in several hundred files.

Firstly, I recorded snapshots of memory in which the player had a constant score while performing actions. The actions involved both changing camera orientation and moving the player to various positions within the world. I performed this procedure over the first three levels, producing a total of about 2000 memory snapshots that represented various levels, camera positions, player locations and clock values. I call these *static state* frames. I analysed the set of static state frames using the Algorithm 2.

Then, I recorded snapshots of memory while the score of the player was increasing. This time, the series of actions involved rolling over coins and fruit in various locations. In total, I gathered roughly 500 memory snapshots that represented various scores. I call these *dynamic state* frames. I analysed the set of dynamic state frames using Algorithm 3.

The outcome of this approach is a single array, h, containing, for each byte index, a set of values that such a byte index can take *while the score is being changed.*

Algorithm 2: Static Analysis

```
Result: A frame containing only values
        for bytes that are constant
$F$ is an array of static frames;
$f_0 \leftarrow$ the first frame in $F$;
while frames in $F > 1$ do
    $f_1 \leftarrow$ the second frame in $F$;
    for byte index $i \leftarrow 0$ to $2M$ do
        if $f_0[i] \neq f_1[i]$ then
            $f_0[i] \leftarrow \perp$
        end
        delete $f_1$ from $F$;
    end
end
return $f_0$;
```

Algorithm 3: Dynamic Analysis

```
Result: A frame containing lists of
        possible score values
Input: $m$ is the output of the static frame
       analysis
$G$ is an array of dynamic frames;
$g_0 \leftarrow$ the first frame in $G$;
for byte index $i \leftarrow 0$ to $2M$ do
    if $m[i] = \perp$ then
        $g_0[i] \leftarrow \perp$
    else
        $g_0[i] \leftarrow \{g_0[i]\}$
end
while frames in $G > 1$ do
    $g_1 \leftarrow$ the second frame in $G$;
    for byte index $i \leftarrow 0$ to $2M$ do
        if $g_0[i] \neq \perp$ then
            $g_0[i] \leftarrow g_0[i] \cup \{g_1[i]\}$
        end
        delete $g_1$ from $G$;
    end
end
return $g_0$;
```

Figure 3.7: The methods used in obtaining the locations of game variables in RAM.

On the assumption that the score was stored as an integer, I created a list of 4-byte word-aligned segments, of which the first byte had a corresponding set in h that had a cardinality of 2 or greater. I chose to consider the first byte as I had found that the PSX's MIPS processor used little-endian byte order [20].

Using this method, I was able to find several satisfying word addresses. I ordered the list by the cardinality of the corresponding sets in h to find the words that changed the most during the score change. I investigated the contents of each possible location manually and in order. This way, I was able to confirm that I had found the correct segment by playing the game and observing the decoded output of that segment. The behaviour of the game was such that score incremented *several times* between the move start and end of the move to collect a coin. Using a similar method, I was able to locate the game variables shown in Table 3.2.

3.2.2 Game Levels

Neither the emulator nor the RL agent has a notion of the current game *level*. In Kula World, players progress to a level by winning the levels below it. It is important that the abstraction support a number of different levels, since just training and testing on one level—especially a simple one—would not be a thorough investigation of the agent's ability. As such, I designed the abstraction to support any of the first ten levels of Kula World through the use of a simple constructor argument.

Type	Address	Description
`int`	`A2E90`	remaining time for level *in units of 0.2s (seconds =* $\frac{value}{50}$*)*
`int`	`A5498`	camera position *this value changes during animations*
`int`	`A3E84`	score
`int`	`A2ADC`	game state *key in Appendix B*
`int`	`A2EA0`	number of levels completed
`int`	`A47A8`	indicator of whether level is finished

Table 3.2: Game variables in memory.

The way that levels are implemented uses PSXRL's `loadState` method. This allows the console to resume its execution to that of a previous register and memory configuration stored on disk. I obtained these configurations by playing the game manually with PSXRL and calling `saveState` at the start of each level.

3.2.3 Performing Actions

The action definitions that are described in the Preparation chapter are relatively simply implemented using PSXPY. Since buttons require both a *depress* and a *release* instruction, I chose to specify that each takes 50ms when the game is in `fast` mode, and 200ms when it is not. This was chosen due to testing. When the delay was too short, moves would sometimes not complete; when the delay was too long, multiple moves would occur instead of one.

3.2.4 State Delimitation

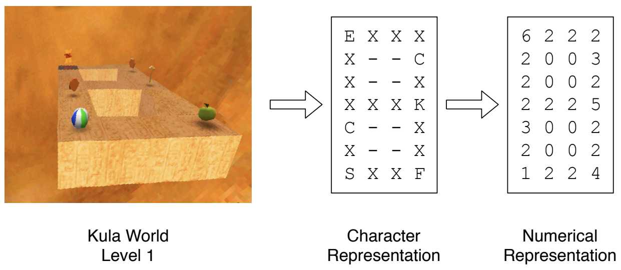
Perhaps the most important role of the abstraction is that it converts Kula World's asynchronous state transitions (as discussed in §2.4) into the synchronous function-return semantics of OpenAI Gym. To do this, the abstraction must be able to tell when a move has finished and calls to `step` must block until this occurs. There are three important ways in which the game indicates that a move is in progress:

- **Camera Position**: The *camera position* game variable updates repeatedly while a move is taking place. Specifically, this represents the camera moving with the player. When the camera position is changing, a move is taking place.
- **Score**: When a player collects a coin, the score increments sequentially before reaching a known value. While the score is still changing, a move is taking place.
- **Audio**: When background music is turned off, it is possible to isolate sound that is played during a player's actions. While the audio is still playing, a move is taking place.

The implementation of this method is simplified heavily by the tools available in PSXPY. It requires just three notification callbacks and a single timer. The timer is set to some *initial delay* value when a move is performed; from this point until the timer being done, any changes to *score*, *camera position* or audio playing will cause the timer to be reset to a special *delay* [2] value. Once the timer expires, the move is considered to have finished.

3.2.5 State Representation

The abstraction was built to support all three types of state representation. For the simple and agent-local representations, the abstraction must track the position of the player as it performs actions. In order to do this reliably, a full model of the level must be constructed and provided to the abstraction. I decided to use an ASCII text file to encode the level map, with specific characters to represent squares. During execution, these characters are converted to numerical values in order to be used as input to the learning network. Figure 3.8 shows how the level map is converted between its ASCII and numerical representations.



```
E X X X
X - - C
X - - X
X X X K
C - - X
X - - X
S X X F
```

```
6 2 2 2
2 0 0 3
2 0 0 2
2 2 2 5
3 0 0 2
2 0 0 2
1 2 2 4
```

Figure 3.8: Translating between the level, an ASCII-encoded level map and a numerical representation. I chose integer values for the numerical representation depending on how desirable a certain position was to the agent. This was mainly chosen to create a linear distinction between values representing areas that are safe for the agent to be in and areas that are not.

I then use the `performAction` method inside of the game abstraction to update position given this map. The position that is stored about an agent is a ternary tuple containing the x position from the origin, y position from the origin and the direction that the agent is facing. These are sufficient since I only encode the first and second levels of Kula World using this method and they are both two dimensional. This method also allows the abstraction to predict when a jump will cause the player to lose the game and end the episode, saving time during training.

Simple State

For the simple state representation, the state is encoded as a single scalar value. This is achieved by a simple mapping of ternary tuples to integer values:

$$(x, y, z) \mapsto 4wy + 4x + z$$

(Where $x, y, z \in N \wedge 0 \leq x < w \wedge 0 \leq z < 4$.)

Since the number of possible directions is 4, this represents a bijection: no two different states are represented by the same number. An example of this numbering in a 3x3 world is shown in Figure 3.9 for each facing direction.

[2]This needs to be chosen correctly in order to delimit moves properly. The choice of this value is discussed in §4.1.3.

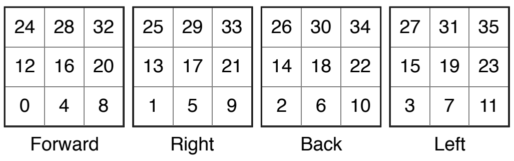


Figure 3.9: Numbering 2D space with a directional component. This example shows a hypothetical 3x3 level with 4 directions.

Agent-Local State

The agent-local representation is composed of a crop and rotation of the level map. Since the cell values do not change when objects are collected, the state does not capture *progression* through a level. To combat this, values for cumulative reward and remaining level time are included in the representation, these additional values constitute *episode state*. Figure 3.10 shows the construction of Agent-Local state using positional information and the level map.

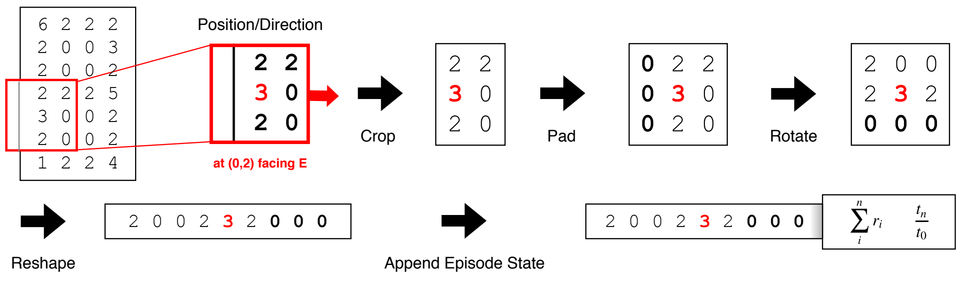


Figure 3.10: Pre-processing state representations for the agent-local state representation.

Complex State

The most simple form of complex state that is used involves the output of the frame buffer. This is obtained through PSXPY's screenshot method and processed using OpenCV to extract cropped frames with separated RGB components, as shown in Figure 3.11.

As part of my extension objectives, I aimed to extract audio from the game in a way that could be used to aid learning. Using the techniques described in §3.1.5, the audio outputs shown in Figure 3.12 can be obtained as moves are performed.

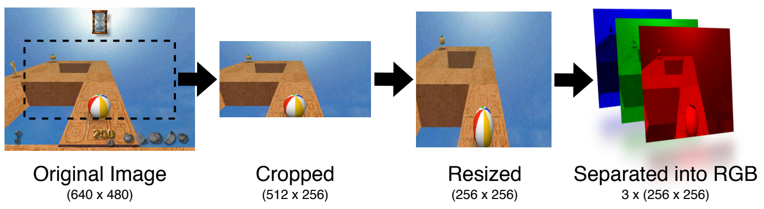


Figure 3.11: Frame processing used for complex state representations. Images have their red, green and blue (RGB) channels separated.

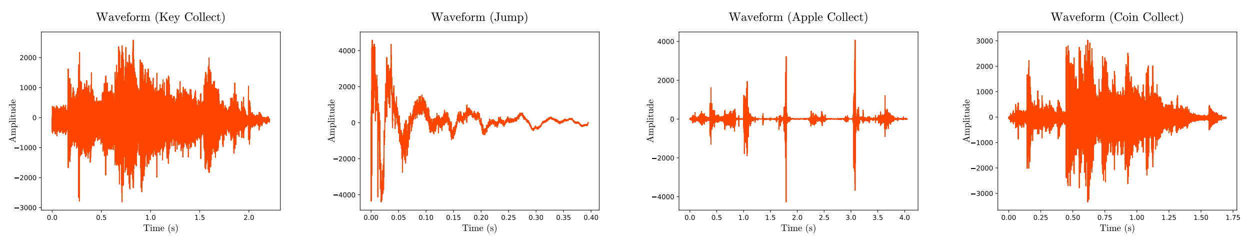


Figure 3.12: Raw audio output for a selection of moves within Kula World, obtained using PSXPY's audio recording feature. The abstraction employs PSXPY's early and late cropping, as well as amplitude termination as described in §2.5, meaning that the output representation from PSXPY is restricted *only* to non-silent audio.

3.2.6 Reward Determination

In order to determine how reward should be allocated to certain moves, I needed to consider both the mechanics of the game and the learning methods that I would be using. In order to aid stability in the learning process, I required that the output of $Q(s|a)$ be strictly between -1 and 1. From the update rule given by Equation 2.8 in §2.3, we have from for some a, i:

$$Q(s|a) = R(s,a) + \gamma \max_{a' \in A} Q(G(s,a)|a').$$

Kula World's countdown timer guarantees that episodes are finite, so for all states s and actions a, there is some $\mathbf{R} = [r_0, r_1, \ldots, r_{n-1}]$ such that:

$$\begin{aligned} Q(s|a) &= r_0 + \gamma\,[r_1 + \gamma\,[\ldots + \gamma r_{n-1}]\ldots] \\ &= \sum_{i=0}^{n-1} \gamma^i r_i \end{aligned}$$

Suppose $r_{\max} = \max(\mathbf{R})$, then we can bound $Q(s|a)$ by:

$$Q(s|a) = \sum_{i=0}^{n-1} \gamma^i r_i < \sum_{i=0}^{n-1} \gamma^i r_{\max} = r_{\max} \sum_{i=0}^{n-1} \gamma^i < r_{\max} \sum_{i=0}^{\infty} \gamma^i = r_{\max} \frac{1}{1-\gamma} \qquad \text{, for } 0 < \gamma < 1 \tag{3.1}$$

So by choosing $r_{\max} = 1 - \gamma$, we have that $\forall a, s.\, Q(s|a) < 1$. Using a similar argument, we can pick some value $r_{\min} = \gamma - 1$ that bounds $\forall a, s.\, Q(s|a) > -1$. Hence, by choosing all rewards r having $\gamma - 1 \leq r \leq 1 - \gamma$, we have that, over any finite number of state transitions:

$$\forall a, s. -1 < Q(s|a) < 1. \tag{3.2}$$

I chose to allocate reward in the way shown in Table 3.3.

3.2.7 Features for Evaluation

Since the abstraction is entirely operated using Python, it is already well-suited to automated testing. I added several more features in order to make this process easier for my evaluation and for users of the library. Users of the library can easily request:

Game Feature	Score Change	Reward
Coin Collect	+250	$\frac{1}{2}(1-\gamma)$
Fruit Collect	+500	$\frac{7}{10}(1-\gamma)$
Key Collect	+1000	$\frac{4}{5}(1-\gamma)$
Win Level	N/A	$1-\gamma$
Lose Level	N/A	$-(1-\gamma)$

Table 3.3: Reward allocation for different increases in score, in terms of discount γ.

- the *real duration* of a move, giving the amount of time that a move took to be performed;
- the *game duration* of a move, giving the amount of time that has elapsed within a game between the start and end of a move;
- a *memory dump* in which the console's RAM is copied to file.

In addition, the console's execution speed and the delay timer can easily be modified, which allowed me to run automated tests to determine which values should be used.

3.2.8 Summary & OpenAI

All of the aforementioned components of the abstraction are designed with the intention of exposing a very simple interface to the agent. I decided that the most appropriate way to do this would be to conform to the OpenAI Gym API in the form of an OpenAI Gym *Environment*.

OpenAI Gym *Environments* expose three functions: `reset`, which restarts the episode and returns the initial state, `step`, which takes an action as an argument and performs it within the environment and `render`, which renders the state to a window or as text.

The `step` function takes an integer that represents an action and returns an `OpenAITuple` representing the outcome of the move. The `OpenAITuple` contains: `state`, which is the value of the state of the system after an action has taken place; `reward`, which gives the reward gained by performing a certain action; `done`, which is a Boolean value representing whether the episode has finished; and `info`, which gives extra information about the environment.

All of the Gym Environment's functions are performed synchronously and return with relevant data. This is where the abstraction's asynchronous to synchronous approach is most useful. While the PSX console executes entirely asynchronously from the agent, the abstraction features that I have described expose an entirely synchronous set of methods.

Each `KulaEnv` holds an instance of `KulaGame`—the abstraction—and each instance of the abstraction holds a PSXPY `Console` instance. This configuration is shown in Figure 3.13. To conform to the OpenAI Gym specification, `KulaEnv` simply executes the appropriate methods from its instance of `KulaGame`. Using the interface, it is relatively simple to write different instances of `KULA_ENV` that have different state representations or reward models.

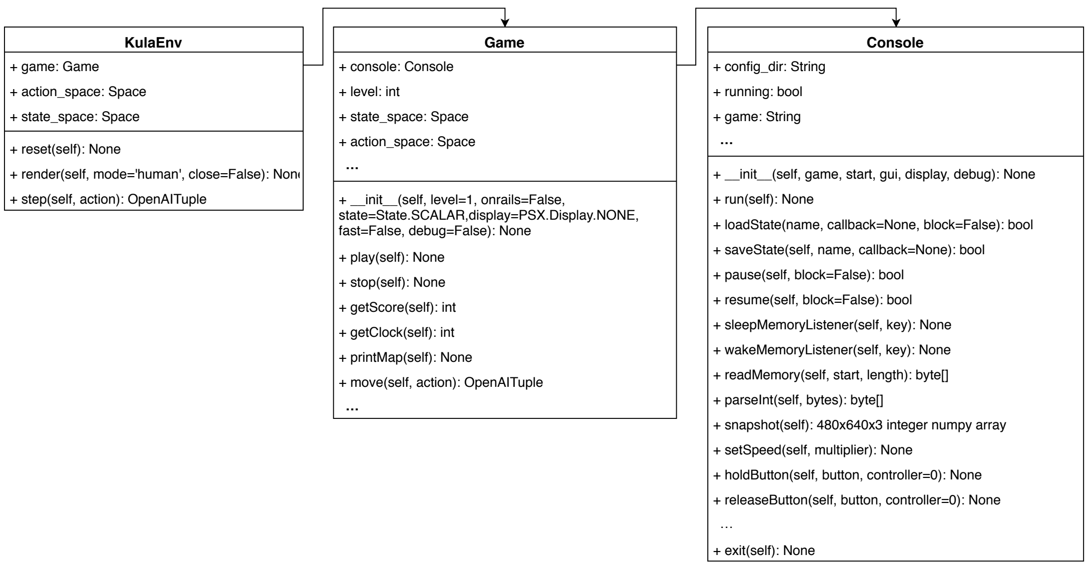

KulaEnv
+ game: Game
+ action_space: Space
+ state_space: Space
+ reset(self): None
+ render(self, mode='human', close=False): Non
+ step(self, action): OpenAITuple
Game
+ console: Console
+ level: int
+ state_space: Space
+ action_space: Space
...
+ __init__(self, level=1, onrails=False, state=State.SCALAR,display=PSX.Display.NONE, fast=False, debug=False): None
+ play(self): None
+ stop(self): None
+ getScore(self): int
+ getClock(self): int
+ printMap(self): None
+ move(self, action): OpenAITuple
...
Console
+ config_dir: String
+ running: bool
+ game: String
...
+ __init__(self, game, start, gui, display, debug): None
+ run(self): None
+ loadState(name, callback=None, block=False): bool
+ saveState(self, name, callback=None): bool
+ pause(self, block=False): bool
+ resume(self, block=False): bool
+ sleepMemoryListener(self, key): None
+ wakeMemoryListener(self, key): None
+ readMemory(self, start, length): byte[]
+ parseInt(self, bytes): byte[]
+ snapshot(self): 480x640x3 integer numpy array
+ setSpeed(self, multiplier): None
+ holdButton(self, button, controller=0): None
+ releaseButton(self, button, controller=0): None
...
+ exit(self): None

Figure 3.13: A UML representation of how each component is connected. This structure makes it easy for modifications to be made and new games or levels to be used.

3.3 Learning

In implementing the learning environment, the abstraction provided by OpenAI Gym negates the need to consider details specific to either the PSX console or to the Kula World game. The implementations of learning using each state representation differ greatly, so they are each described separately. In all cases, actions are represented as an integer from 0 to 3 for forward, right, left and jump forward, respectively.

The simple method uses tabular Q-Learning, by storing the state s in a table along with the estimated value for $Q(s|a)$ for each action a. The latter two methods used a modification known as *Deep Q-Learning* (DQN) in which the value for $Q(s|a)$ is estimated using a deep neural network[3]. This works by employing a neural network to learn parameters θ subject to the update rule:

$$\mathbf{Q}^{\theta}(s)[a] \leftarrow r + \gamma \mathbf{Q}^{\theta}_{\text{best}}(s'). \tag{3.3}$$

This update is applied for every action which transitions the game from state s to s' with reward r. $\mathbf{Q}^{\theta}_{\text{best}}(s')$ represents the highest value in the output of $\mathbf{Q}^{\theta}(s')$.

3.3.1 Exploring the World

In order to produce a good approximation of $Q(s|a)$ for all s and a, the agent must visit a large number of states. Simply visiting these states randomly is a possible method of doing this, however, in Kula World, this would lead to the player jumping off the platform and losing the game with very high frequency. The aim to cover a large number of distinct states would benefit from the agent choosing actions based on its learned knowledge–so as not to jump off the platform repeatedly. If the agent follows its knowledge only, it would not learn anything since the information it receives would only serve to reinforce whichever set of actions its policy

[3]This is because the state space would be too large to represent in a tabular fashion.

told it to. A popular solution to this is known as the ϵ-greedy strategy [13], which has agents follow their policy in some cases and pick randomly in others. The choice to act randomly is made with some non-zero probability ϵ, which decreases as the agent learns more.

All of the techniques in this project employ the ϵ-greedy strategy in various ways. In general, they will start with $\epsilon = 0$—entirely random behaviour—and take an integer number of moves m as an argument, over which to explore. During the exploration stage, ϵ will decrease linearly; after it, ϵ remains constant at a non-zero value.

3.3.2 Game Memory

For DQN, I implemented *experience replay*. An *experience* is a tuple (s, a, r, s') describing an initial state (s), an action taken (a), the reward achieved as a result of the action (r) and the resulting state (s'). Experience replay is a method in which agents can retrain on past *experiences* that they have had. This helps to achieve two important goals:

- Ensuring that the agent doesn't 'forget' about earlier parts of a level as it learns, due to seeing an increase in experiences from later in the level.
- Increasing the amount of training data available to train the network. Reusing old experiences to increase the amount of training data is valid in this case since levels begin in a deterministic way and so experiences will not differ greatly between episodes for a given set of actions.

The latter advantage is specifically important given that the agent must wait for asynchronous transitions to occur on the console, meaning new training data takes a while to collect. While agents typically pick a random set of past experiences to train on, other methods for choosing experiences exist [19] which are not discussed here.

3.3.3 Implementing Simple Learning

Q-Table

The Q-Table holds a mapping between pairs of a state s and an action a and the corresponding Q value $Q(s|a)$. States that correspond to positions that the agent cannot exist in (or would lose the game in attempting to) are redundant; there are 8 such positions for the first level, leading to 32 redundant states. This means that 128 state-action pairs exist that will never be updated in the Q-table. To ensure that this does not result in wasting memory for the agent, I use a Python dictionary to store the table; this approach maintains fast look-up without needing to allocate memory for redundant cells.

Implementation

The implementation of the simple case took place only over the first two levels of Kula World, as these were the only ones that could be expressed totally within a 2D level map like those in Figure 3.8.

3.3.4 Implementing Agent-Local Learning

Neural Network Architecture

I used a simple fully-connected network with 5 layers with an input layer size of 11 and the output layer size of 4 [4]. The network uses the PReLU rectifier function for each hidden layer:

$$\mathrm{PReLU}(x) = \begin{cases} ax & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

(Where $a \in \mathbb{R}$ is a learned parameter.)

For the output layer, I use the hyperbolic tangent (tanh) as the activation function. This transforms all network outputs into the range $[-1, 1]$ and improves training stability[5].

Learning Procedure

The general *learning procedure* for the agent-local case is shown in Figure 3.14. The network is trained according to the update rule given by Equation 3.3.

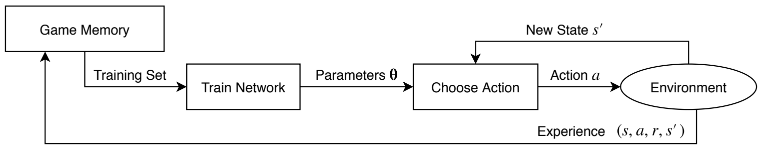


Figure 3.14: The procedure used for Deep-Q learning.

3.3.5 Implementing Complex Learning

State Interpretation

Complex learning is unique compared to the other two methods in that it contains multiple distinct representations of state. Visual data, for example, has no clear structure in common with *episode state*. As a result, simply concatenating each representation and feeding it to a single Q-Network, as was performed in the Agent-Local implementation, would not suffice.

Visual Input

The most common way of interpreting visual input data in the form of smaller and more meaningful representations is through the use of Convolutional Neural Networks (CNNs). These can take large amounts of pixel-level data and output a single vector that represents some feature or property of the image according to how it was trained.

Simply representing the Q-Network as a CNN would make it hard to incorporate episode state and audio data as inputs. The result of this was to design a new architecture for multi-modal learning. Here, multiple types of state are processed by their own *Interpretive Networks*[6], the outputs from which are then concatenated to produce a linear input to the Q-Network.

[4] This can be found in Appendix C

[5] This decision is examined more carefully in the evaluation.

[6] In the case of this project, these are *disjoint* from one another, but they need not be.

The network used for episode state was fully connected with three inputs, one hidden layer and an output layer of size 6. This network can be found in Appendix C. For interpreting visual data, I followed the guidance of prior work [15] and designed the network shown in Figure 3.15.

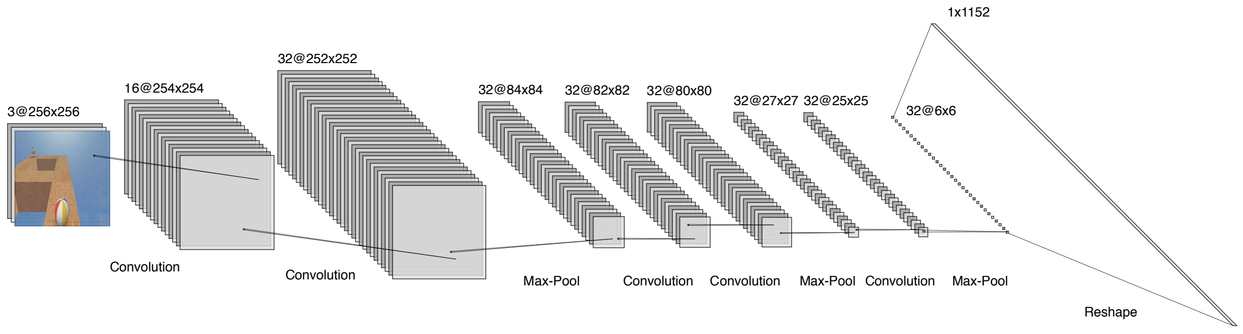


Figure 3.15: The *Interpretive* network for visual state. It is a convolutional network with 1152 outputs. This value is obtained as a result of flattening the final layer of 32 6x6 patches.

Psychoacoustics

While the generally accepted [5] range of normal human hearing is between 20Hz and 20kHz, humans' perception of *pitch* does not vary linearly on this scale. Through experimentation, it has been widely concluded that the relationship between frequency and perceived pitch is close to logarithmic[7]. As such, the mel (**mel**ody) scale represents audio in units of *mel frequency* rather than in Hertz. A common transformation of a frequency f to mel frequency is:

$$\text{mel}(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{3.4}$$

It is possible to consider coefficients that represent the degree to which specific mel frequencies contribute to a sample of audio [18], in much the same way that this is possible with Fourier analysis. These are known as *Mel Frequency Cepstral Coefficients* (MFCCs). The mel scale is split equally into a number of sections equal to the number of required output coefficients. The sections are then scaled according to the relationship in Equation 3.4 so that they correspond to the Fourier domain. Performing a Fourier transform on the input audio allows for coefficient values to be determined. This procedure is used for small overlapping *frames* of the audio input. I separated the input sounds into 25ms frames, used a step of 11.6ms between the start of each frame and derive 13 coefficients [8]. MFCC transformations are shown in Figure 3.16.

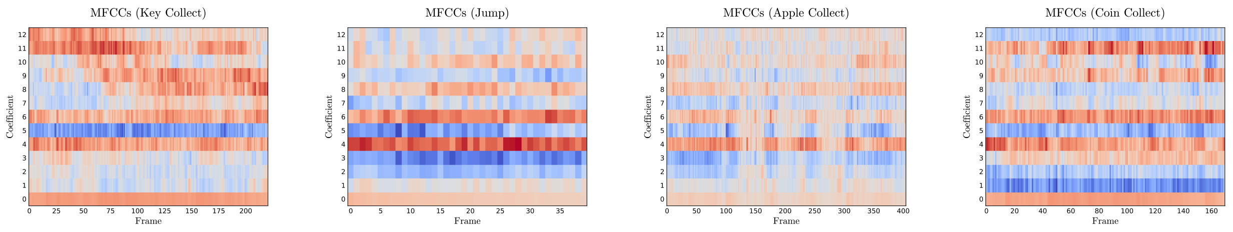


Figure 3.16: MFCCs derived from the waveforms in Figure 3.12.

[7]This relationship exists in most human musical scales. In western musical theory, the scale of semitones can be defined as a logarithmic function of frequency.

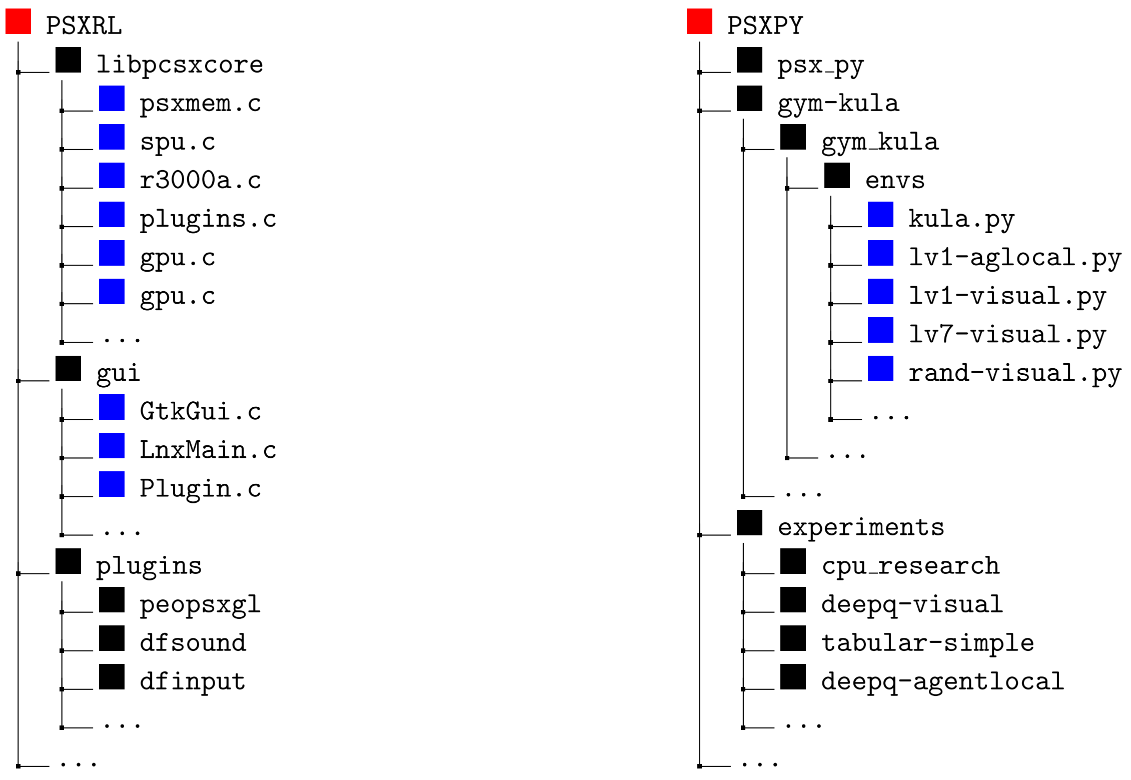
[8]These values are typical [6] for applications of MFCCs.

3.4 Professional Practice

Since my work utilised a commercial game, Kula World, it was important to conform to relevant copyright and licensing restrictions. I did not make any modifications to the game, I employed it for strictly non-commercial (research) use and I did not copy or distribute the game binary along with my source code.

I also made heavy use of the PCSXR emulator, which is released under a *GNU* public licence. Under these terms, I have the "legal permission to copy, distribute and/or modify" [7] the source code. Additionally, the terms dictate that any derivative work that I produce will also fall under the same GNU licence terms.

3.5 Repository Overview



```
🟥 PSXRL
├── ⬛ libpcsxcore
│   ├── 🟦 psxmem.c
│   ├── 🟦 spu.c
│   ├── 🟦 r3000a.c
│   ├── 🟦 plugins.c
│   ├── 🟦 gpu.c
│   ├── 🟦 gpu.c
│   └── ...
├── ⬛ gui
│   ├── 🟦 GtkGui.c
│   ├── 🟦 LnxMain.c
│   ├── 🟦 Plugin.c
│   └── ...
├── ⬛ plugins
│   ├── ⬛ peopsxgl
│   ├── ⬛ dfsound
│   ├── ⬛ dfinput
│   └── ...
└── ...
```

(a) The directory structure for PSXRL. This follows the same structure as PCSXR, separating plugin, emulation and user interface code. The `plugins` folder contains `FakeJoy` and GPU/SPU modifications. Most `procPipe` methods were added in `gui`. Modifications such as memory listeners and state loading were made in `libpcsxcore`.

```
🟥 PSXPY
├── ⬛ psx_py
├── ⬛ gym-kula
│   ├── ⬛ gym_kula
│   │   ├── ⬛ envs
│   │   │   ├── 🟦 kula.py
│   │   │   ├── 🟦 lv1-aglocal.py
│   │   │   ├── 🟦 lv1-visual.py
│   │   │   ├── 🟦 lv7-visual.py
│   │   │   ├── 🟦 rand-visual.py
│   │   │   └── ...
│   │   └── ...
│   └── ...
├── ⬛ experiments
│   ├── ⬛ cpu_research
│   ├── ⬛ deepq-visual
│   ├── ⬛ tabular-simple
│   ├── ⬛ deepq-agentlocal
│   └── ...
└── ...
```

(b) The directory structure for all of the relevant Python scripts. This includes PSXPY as well as the abstraction, experiments and evaluations. Everything in these directories was written by me. Most notably, `kula.py` constitutes most of what is referred to as the *abstraction* in this dissertation.

Figure 3.17: A repository overview for the code submission.

Chapter 4

Evaluation

The outcome of this project is one that has never before been built or evaluated. As a result, it is not possible to perform an evaluation of the relative merits of my work compared to that of others. Instead, I carry out tests that measure the outcome of my work against my success criteria, justify some of the choices made in the implementation and attempt to conclude whether Deep Q-Learning is a suitable approach within the more complex environment of PlayStation games. This section of the dissertation is bifurcated into the principal evaluations: that of the primary work product and that of learning approaches within this environment.

4.1 The Environment

4.1.1 Core Functionality

The environment satisfies all requirements listed in §2.6. I verified support for Ubuntu by running a single PSXRL instance of Kula World and running a sequence of actions that: begin a new game, take a snapshot and close the console. If this sequence both finished without throwing an exception *and* resulted in a visually identical snapshot as would be obtained manually, they were considered to have passed. There were no failures among the configurations that I tested:

- Ubuntu 16.04 with Unity running in a virtual machine
- Ubuntu 18.04 with Unity running in a virtual machine
- Ubuntu 16.04 with LXDE (Lubuntu) running in a virtual machine
- Ubuntu 16.04 Server running on a remote machine
- Ubuntu 18.04 Server running on a remote machine

4.1.2 Concurrency

Requirement 7 in §2.6 specified that the emulator should support multiple concurrently executing instances. Since several forms of IPC are employed, it is important to ensure that there are no cases in which the two instances of the emulator attempt to open the same resource, pipe or shared memory. Additionally, it is important that it is *feasible* to run multiple instances at once and that doing so does not substantially inhibit the performance of each instance.

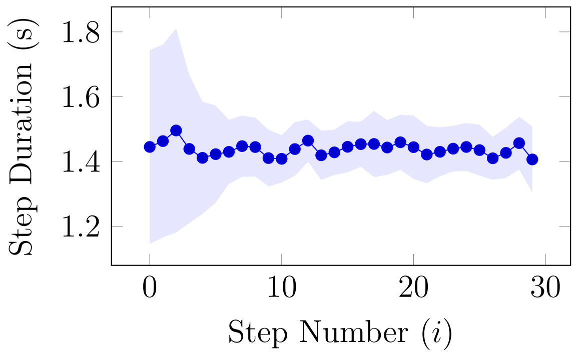
Methodology

To investigate this, I chose a test environment typical of one that may perform such machine learning. I used a Virtual Private Server (VPS) from DigitalOcean with a new install of Ubuntu 16.04, 32GB of RAM and 16 vCPUs. I then wrote a script to run the Kula World Gym Environment, perform the 'turn right' move 30 times and record the *step duration* for each move. Specifically, this records the amount of time that elapses between the environment's `step` method being called and it returning.

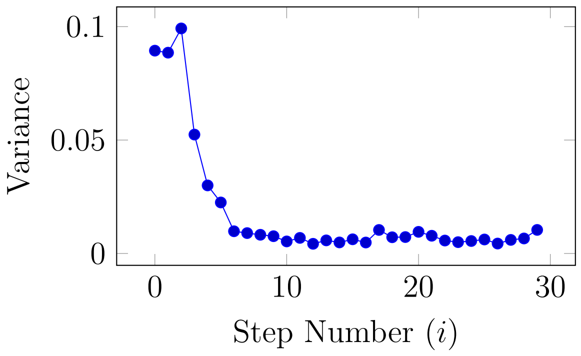
I then wrote a bash script which concurrently starts a specified number of instances of the test script in separate `tmux` sessions, along with a single script that records the system's CPU utilisation and RAM usage for a period of 30 seconds. For the analysis below, I chose to run the script for multiples of 5 between 5 and 55 along with 1 and 58 (a hard-coded maximum in PSXPY[1]).

Analysis

As shown in Figure 4.1, the value of step duration appears to stabilise after around 8 moves. The discordant period prior to this can be visualised in Figure 4.1a, where the light blue area represents values within a single standard deviation of the mean. This is likely due to some steps being taken as other environments are launching (leading to an increase in step time) or before other environments have launched (leading to faster step times). As a result, I chose to ignore the first 8 steps for my analysis. A plot of the mean step duration for each agent against the number of concurrently executing PSXPY instances is shown in Figure 4.2a.



(a) Average Step Duration for $n = 58$.



(b) Average variance of step duration.

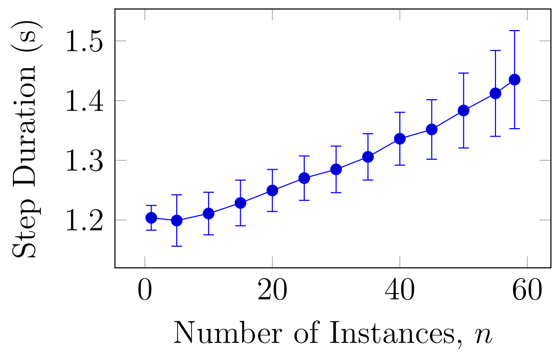
Figure 4.1: A visualisation of variance within the step duration data. On the right, I plotted the *mean* of the variance values over all n for each step. Measurements appear to stabilise after the eighth step.

It is clear that the step duration is relatively sensitive to an increase in concurrent instances. In order to put this in a more meaningful context, it is useful to consider how many moves can be carried out per second. This value is proportional to the rate of collection of experiences: if it grows slowly, parallelism in any form will not have any impact on learning speed; whereas if it grows quickly, there may be an efficiency advantage in using a parallelised algorithm such as A3C [14]. For some mean move duration d and number of instances n, the total number of moves per second, M, given by:

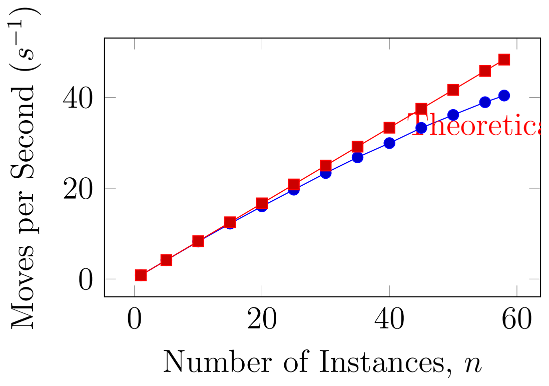
$$M(n, d) = \frac{n}{d}$$

A plot of this is shown in Figure 4.2b, compared to the theoretical best case in which step duration does not reduce with more instances. The outcome is very promising, with the measured performance being very close to the theoretical best and suggesting that an implementation of A3C would represent an increase in learning efficiency over single-threaded approaches.

[1]This restriction comes from protocol allocating only one byte for the instance ID.

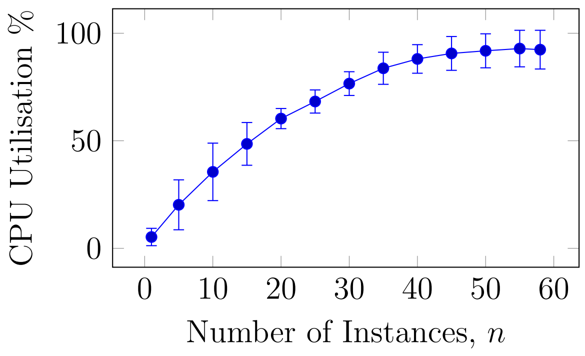


(a) Step duration appears to grow linearly with the number of instances.

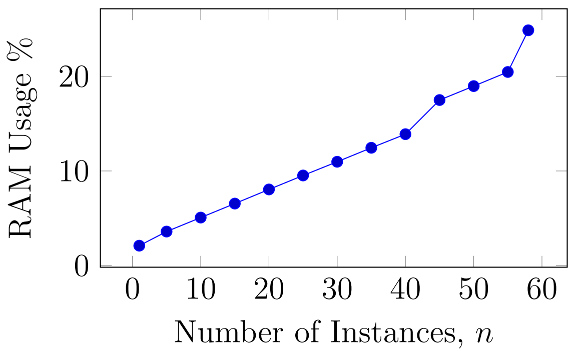


(b) The number of steps carried out per second, compared to the theoretical best-case.

It was important to also consider the effect on the CPU and RAM since, if either of these resources was used too heavily, the environment might not run properly. Furthermore, machine learning research is usually carried out on *shared machines*, leading to resources being heavily restricted. The results can be seen in Figure 4.3a and Figure 4.3b. These results were obtained through the use of the same procedure as for the concurrency analysis above.



(a) CPU utilisation as the number of instances increase.



(b) RAM usage as the number of instances increase.

Figure 4.3: CPU utilisation appears almost linear for the first 15 instances, before growing more slowly as the system reaches capacity. One standard deviation is indicated in the error bars, showing how much the CPU usage varied during the duration of the test. RM usage appears to increase linearly. No error is indicated here as the variation was negligible during the test.

These results show an expected linear relationship between n and RAM usage; with the usage staying relatively low throughout. The results imply that memory will almost certainly not be a barrier to performance on machines of this standard. Similarly, the CPU utilisation adapts to what utility is available, while suggesting that similarly capable machines should have sufficient capacity to run the environment with many instances.

4.1.3 Choice of Timer Delay

In order to delimit states, a timer delay has to be chosen as described in Section §3.2.4. If the delay is too short, it may end the move too early. In such a case, any score increase will not be properly measured and the agent may not even be able to complete its next move[2]. In order to measure the *effectiveness* of certain timer values, I built a simple experimental setup. The steps are:

- Instantiate a Kula World instance at level 1
- Override the default timer value to the one being tested and disable *score smoothing*[3]
- Call `step` to make the agent *jump forwards*
- Call `getScore()`. If the value returned is *not* 250, the test has failed.

This test works because jumping forwards at the start of level 1 should yield a score of 250; however, if the timer finished the move too early, the score will *still be increasing* to the correct value. The results of this analysis are given in Figure 4.4. Clearly, larger values for the delay

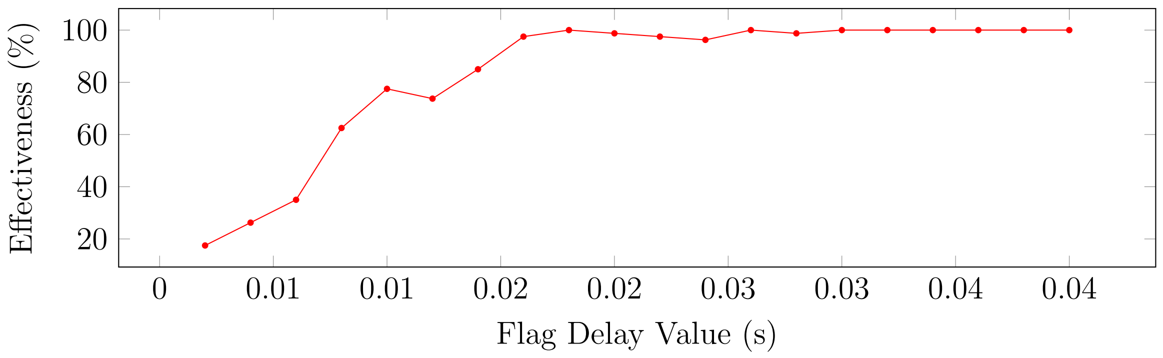


Figure 4.4: Delay values measured by the ratio of moves that they successfully delimit. Moves are successfully delimited if the console runs for long enough to finish the move fully.

will be more successful in checking if moves have finished; however, increasing delay comes at the cost of extending move lengths and thus overall training time. To investigate this trade-off, I measured the length of times that a move takes to be performed, with different delay values. Figure 4.5 shows the effect that increasing timer values have on move duration. I only included *successful* moves, since moves that terminate early will have an artificially low duration. Based on this analysis, I concluded that the timer value of 0.06 was sufficient. It causes a negligible increase in move duration compared to lower values, yet it appears to be entirely successful at delimiting states.

4.2 Reinforcement Learning

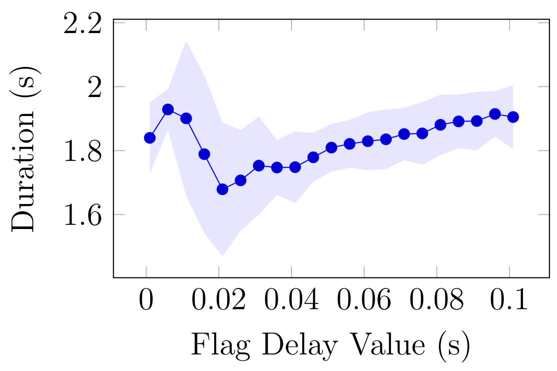
This section details a comparison of the RL approaches that I implemented, along with three parameter evaluations that I performed, finishing with a discussion of generalisation.

4.2.1 Methodology

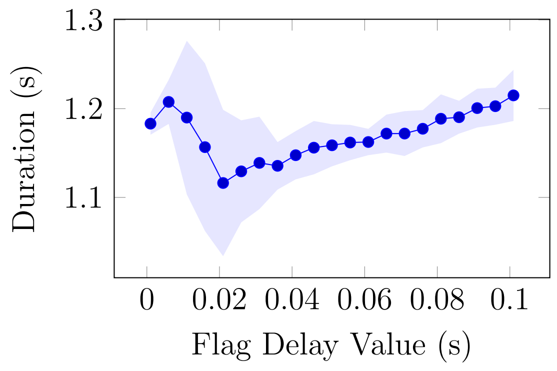
While training the DQN models in this evaluation, I use `pyTorch`. Specifically, I use the *Adam* optimiser and the *Mean-Squared Error* (MS) loss function. Weights are initialised according to a Normal distribution with a mean of 0 and a variance of 0.0001. The *learning rate* parameter

[2]Moves can only occur when the previous one has finished.

[3]This is a mechanism that attempts to artificially round the score up to the nearest 'possible' score in the case of a fault. In this case, we want to expose those situations.



(a) Measuring *game-time* duration.



(b) Measuring *real-time* duration.

Figure 4.5: Figures showing the relationship between the delay value and move duration. Figure 4.5a shows the amount of time that has elapsed within the game, as compared to Figure 4.5a which shows the amount of time that has elapsed for the agent. The former impacts the number of moves an agent can make in a game, whereas the latter impacts how long it takes the agent to observe moves. For values of delay above 0.04, both game and real duration measurements vary linearly. This is to be expected since longer delay values mean that moves take longer to be recognised as finishing.

was set at 0.0025 for agent-local models and 0.004 for complex models. I used a `GameMemory` size of 5000 and picked 80 as the maximum size of each training batch. Each training batch was trained for 15 epochs. The ϵ-greedy strategy was employed by having ϵ decrease linearly from 1.0 to 0.1 over 10,000 *moves*.

In general, training occurs by splitting episodes into *training* and *testing* groups. While training episodes employ the ϵ-greedy strategy detailed in §3.3.1 to pick moves at random with a probability of ϵ, testing episodes always choose the move that yields the greatest Q value. As such, the testing episodes serve to demonstrate the true ability of an agent to play a level. For this evaluation, the agent performs a testing episode after nine training episodes.

4.2.2 Comparison of Approaches

I trained each of the approaches on level 1 for 1000 episodes. I chose to include a random agent in order to confirm that the learning algorithms perform better than random chance. The results are shown in Figure 4.6. The proficiency of each learning approach is almost indistinguishable and greatly superior to random chance. Each agent achieved the top score of 4000 at least once during testing and all appeared to converge at a score of either 4000 or 3750. Since the agent-local and simple state representations were far more explicit than that of the visual approach, this result appears to confirm that the visual state representation can nonetheless be parsed into an equally useful representation. This is very promising, as visual state representations are far more versatile.

4.2.3 Choice of Discount (γ)

In order to evaluate my choice of discount value, I trained an agent with over 20000 moves on level 2 using both 0.8 and 0.9 as values for γ. I did not choose to use any additional values

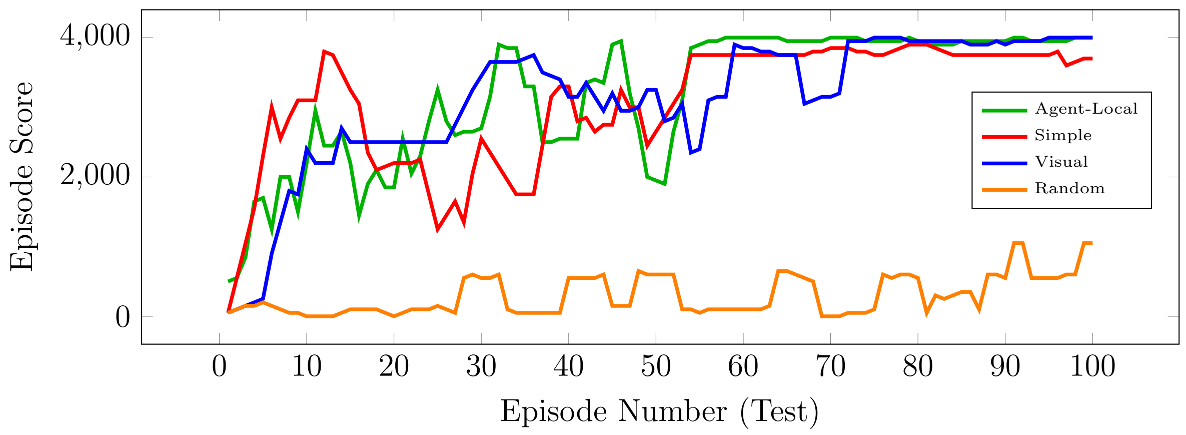


Figure 4.6: The score for agents trained on each of the methods. Each agent was trained for 1000 episodes. The values shown represent the moving average total reward over 5 episodes.

since the use of 0.99—common for DQN [15]—caused the network to saturate at values of Q at 1 or -1 and values that are smaller than 0.8 would have been too short-sighted. Figure 4.7 shows training and testing rewards for both values.

Analysis

The values of reward shown for $\gamma = 0.9$ have been doubled, to make the results comparable [4]. In these results, $\gamma = 0.8$ converges at a lower value of reward than $\gamma = 0.9$. I believe that this is due to the latter case being able to take account of rewards *further into* the level and take actions to reach them. It is worth adding that the latter approach appears to take longer to converge and does so less convincingly.

I also found that picking $\gamma = 0.9$ appears to improve the convergence characteristics of the Q-Network. The value of the *loss function* as the model trains with both values is shown in Figure 4.8.

In my evaluation, $\gamma = 0.9$ represents a better choice than $\gamma = 0.8$ for level 2.

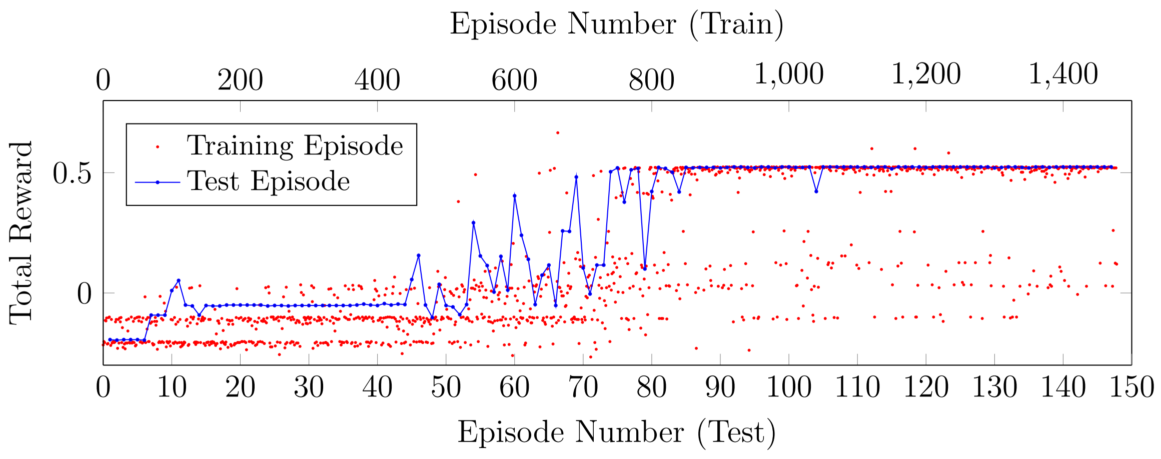
4.2.4 Choice of Epsilon Strategy

In researching this project I came across a minor modification to the ϵ-greedy strategy outlined in §3.3.1. Instead of picking a move *uniformly* at random, the modification picks an action a in state s with a probability proportional to:

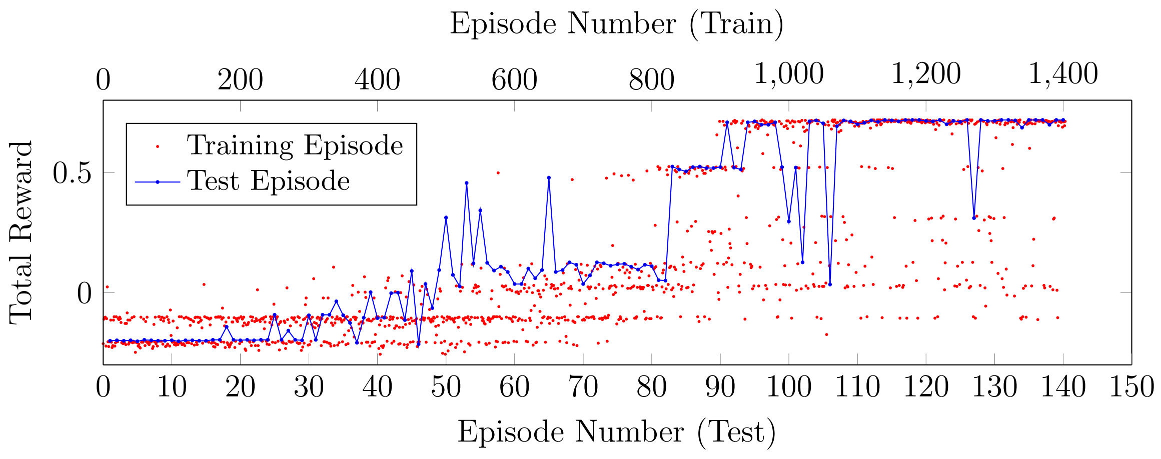
$$K(a) = Q(s|a) + \text{rand}(-0.5, 0.5) \times \text{range}_{a'} \left[Q(s|a') \right]. \tag{4.1}$$

The function allows the agent, even when picking 'randomly', to pick moves that are considered to be more valuable. I ran testing for this on level 7, with the same parameters as above. The results are summarised in Figure 4.9.

[4]This is because reward values are scaled by γ when they are achieved.



(a) A plot of reward for $\gamma = 0.8$. The blue line represents the reward values for test episodes, whereas each red dot represents a training episode.



(b) A plot of reward for $\gamma = 0.9$. Importantly, the reward values shown are *doubled*, since each individual reward was halved in order to bound the value of Q between -1 and 1. Doubling these values makes them comparable with Figure 4.7a. It can be seen that this approach reaches a higher reward at convergence, although experiences much more fluctuation beforehand and converges later.

Figure 4.7: A comparison of training results for different values of γ.

Analysis

My results indicate that the modification provides a small advantage. While both models appeared to converge towards the same reward value, the approach utilising the modification does so more rapidly. This is likely due to the agent training on more experiences of higher reward and thus discovering desirable routes more quickly. While these results are promising, the data that I collected is not clear enough to be conclusive. The alternative approach was not used in any other cases.

4.2.5 Choice of Activation Function

In the Implementation, I described a method through which I constrained the quality value for any possible state to a value between -1 and 1. I then leveraged this in my network through the use of the `tahn` activation function on the output layer, a function that limits the output of

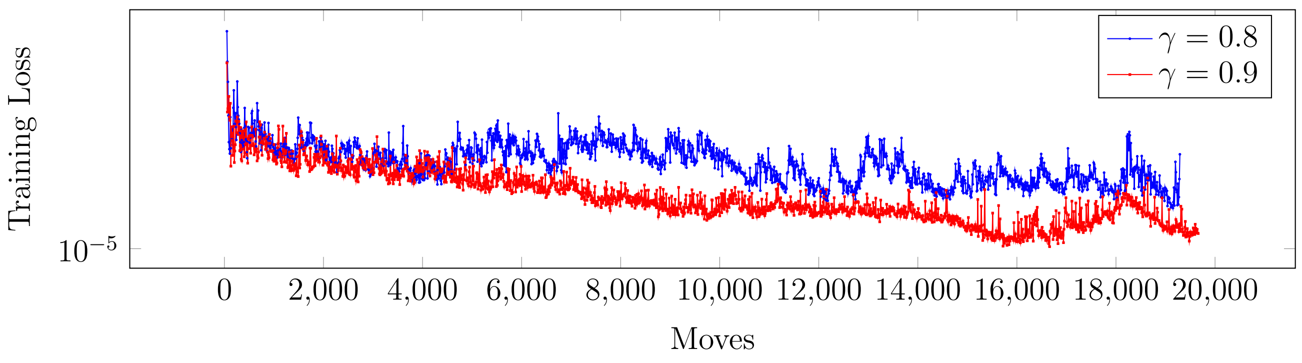
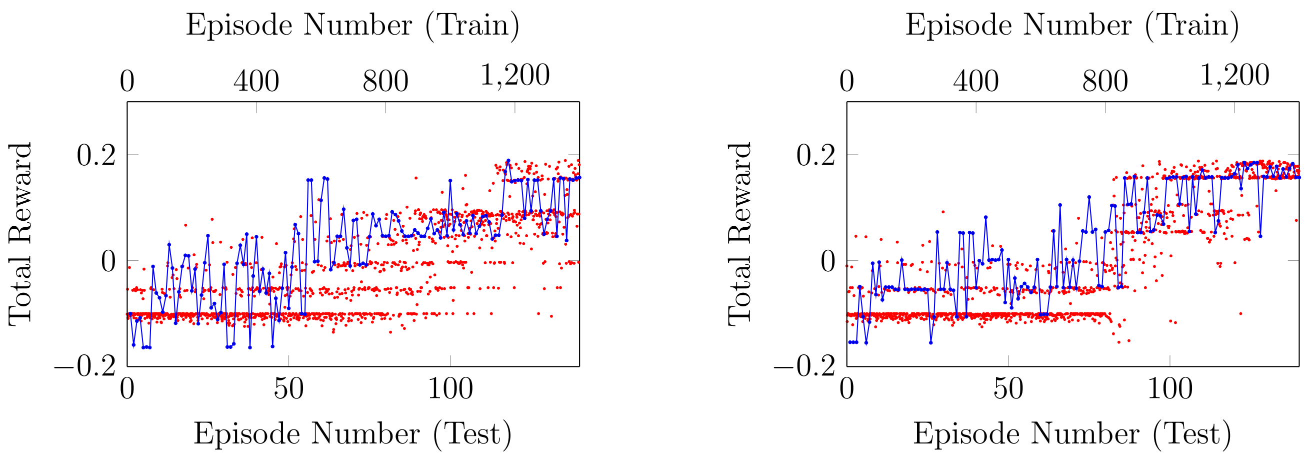


Figure 4.8: A comparison of training loss for different values of γ.



(a) A plot of reward for level 7 where moves are picked randomly with uniform probability.

(b) A plot of reward for level 7 using the modified strategy.

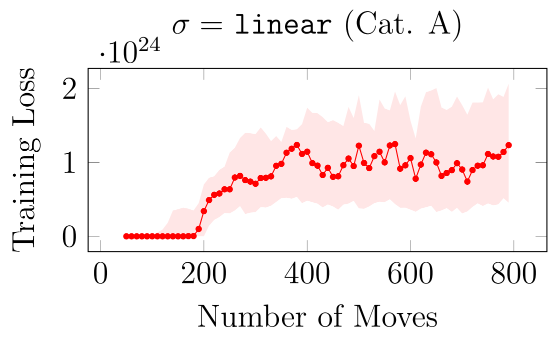
Figure 4.9: Plots of reward for both ϵ strategies.

my network to values between -1 and 1. I claimed that this was chosen as a means to improve learning stability. This claim was made as the result of extensive testing of activation function choices. To collect the following data, I ran 66 concurrent training instances. The instances were split into two categories equally: those using a linear activation function $\sigma(x) = x$ for the output layer; and those using $\sigma(x) = tanh(x)$ activation function for the output layer. All of the instances were training on Kula World on level 1, using the agent-local representation.

Out of 33 experiments, 8 of the networks learning with the linear activation function diverged, whereas all of the networks using the hyperbolic tangent activation converged. The results are summarised in Figure 4.10.

Analysis

These results show that the use of the hyperbolic tangent as the activation function is very important in seeking convergence. Even in those cases in which the linear activation function was used and convergence took place, this is notably slower than with the hyperbolic tangent.



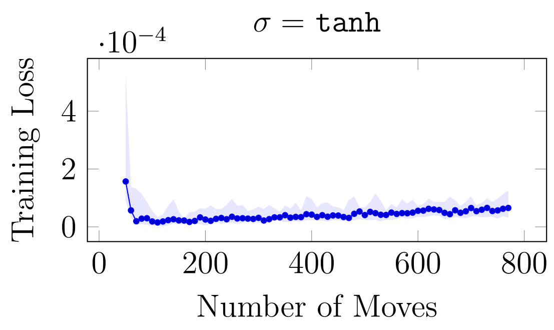
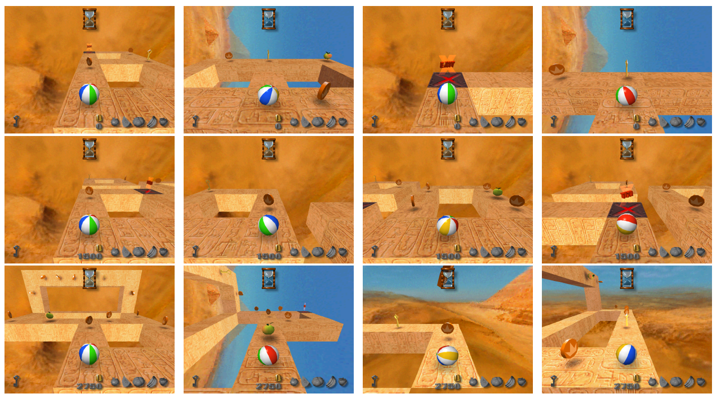


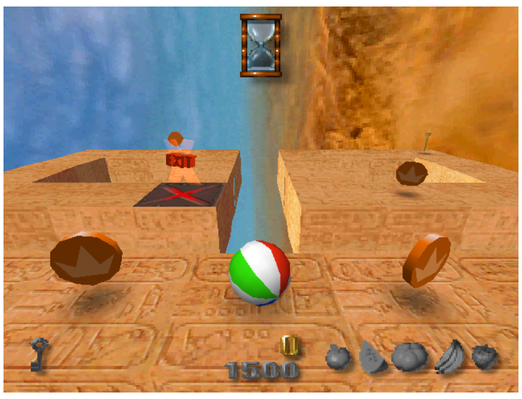
Figure 4.10: Comparison between linear and `tanh` activation functions. Since the variance among results was quite high, I plotted the *worst-performing* cases for each activation function. Of the 33 trained networks, these were chosen to be the 10 cases that exhibited the *greatest value* for loss after training for 700 steps. The difference is mean loss between the worst cases for each function is about 28 orders of magnitude.

4.2.6 Generalisation

Finally, I wanted to evaluate to what extent an agent could be trained such that it was able to play episodes that it had never seen before. I trained an agent, using the visual representation, using 12 different starting positions across the first three levels of Kula World. I chose locations, shown in Figure 4.11a, to have sufficient variability so that the network does not simply *remember* the strategy for the levels that it sees. In order to evaluate the agent, I created the additional location from level 2, shown in Figure 4.11b. This state is presented to the agent during testing episodes and is *never* used as the initial state for a training episode. I was able to set up this evaluation by moving the player (without picking up objects) to each desired location, polling the in-game clock value until there were 80 seconds remaining and using the `saveState` function of PSXPY to capture the console state to disk.



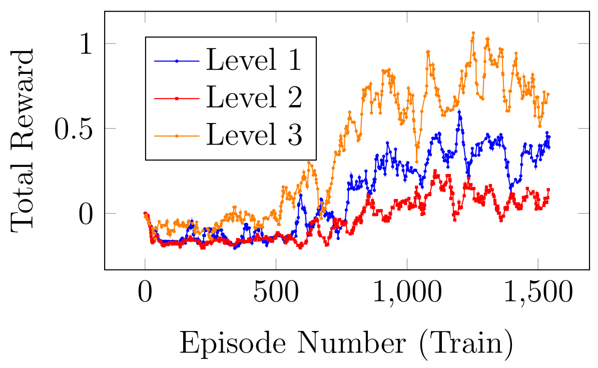
(a) Starting locations, as used in training.



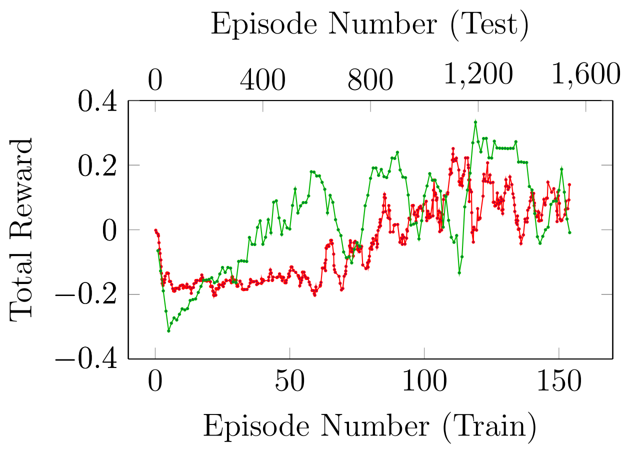
(b) The evaluation episode start location.

Figure 4.11: Screenshots from Kula World showing different training start locations and the designated test location.

The results of this training, separated by levels, is shown in Figure 4.12.



(a) Agent proficiency in playing from the training locations. The line shows a moving average over 10 training episodes.



(b) The green line shows a moving average over 5 testing episodes for the unseen location, the red line is copied from Figure 4.12a.

Figure 4.12: Evaluation of the ability for an agent to generalise.

Analysis

The most surprising conclusion from this evaluation was that level 3 trained to the highest standard, despite it being the most complex level. In fact, level 3 trained to a level sufficient to win the game, whereas level 2 did not. This may be because the visual aspects of level 3 allow different states to be more easily differentiated. As shown in Figure 4.11a, each of the start states for level 2 are predominantly orange or beige in colour, whereas for level 3, they all contain a mix of colours. Since the agent is judging the state solely of the visual output of the console, it may be the case that level 2 appears more visually homogeneous than level 3.

Figure 4.12b shows that the agent is able to generalise some of its learning to the unseen state. However, since the policy for the unseen state is never reinforced through training, it does not appear to converge on a solution. The apparent periodicity of the cumulative reward would suggest that performance in unseen-state episodes is more unstable than in training episodes.

4.3 Summary

Overall, the evaluation shows that the project has been a success. I have evaluated the performance of PSXPY under normal conditions and justified some of the design decisions taken in the implementation. I have also shown that RL algorithms can be used to successfully train an agent to play a Kula World—the first time this has been done for a PlayStation game.

Chapter 5

Conclusion

This project has been a success, resulting in the development of the first RL environment to support PSX games, PSXPY. For the very first time, I also successfully trained an agent on a PSX game and evaluated its performance. The results are very promising, demonstrating a Q-Learning method that is able to converge on a strategy to successfully play Kula World. These results are important because they show how the strategies employed in playing Atari games, such as [15], can be extended to more complex state environments like the PSX.

I was not only able to achieve all of the points in my success criteria, but also both of my extension objectives: I wrote an OpenAI Gym environment that outputs MFCC values along with visual and episode state and I trained an agent on episodes without any hard-coded level information. The only change that I made to my success criteria was that I used the `pyTorch` machine learning library rather than `TensorFlow` and, as a result, I wrote my own DQN implementation rather than using that of *OpenAI Baselines*.

During my work, there were several further extensions that I did not have time to explore:

- Since the agent usually starts from a specified state, the states that it sees are heavily concentrated early in the gameplay of the level. *Prioritised experience replay* [19] attempts to combat this by replaying state transitions that the agent learned the most from. I propose a modification, *prioritised state replay*, in which an agent can *resume play* from a previous state, allowing it to perform new actions in those states. This is easily implemented using the `saveState` and `loadState` methods in PSXPY.
- It may be useful to research methods by which an agent that has already been trained can be used to quickly retrain on an unseen episode. This *marginal specialisation* could have applications in other areas of RL where the amount of training data is small.
- In this project, I propose a method of using *interpretive networks* to process different state information independently. Implementing *feature swapping* between these networks may serve to better correlate different types of data and produce better results.
- In Kula World, the reward achieved by an agent sometimes depends on its past actions. To better encode these dependencies in training, I built a *Recurrent Neural Network* (RNN) through which to predict Q values from a number of previous states. I was not able to perform a hyperparameter search due to the time constraints of the project [1].

I plan to release PSXPY as an open-source project, so that developers and researchers can use it to train and evaluate their RL algorithms on its almost 8000 supported games. I hope that PSXPY will be a useful contribution to the field of RL research and that it will help to inform the design and development of future learning approaches.

[1]The code for this is given in the code submission for this dissertation.

Bibliography

[1] I. Arel et al. “Reinforcement learning-based multi-agent system for network traffic signal control”. In: *IET Intelligent Transport Systems* 4.2 (June 2010), pp. 128–135. ISSN: 1751-956X. DOI: `10.1049/iet-its.2009.0070`.

[2] Dr Tristan Behrens. *Deep Reinforcement Learning and Autonomous Driving*. URL: `https://ai-guru.de/deep-reinforcement-learning-and-autonomous-driving/` (visited on 03/04/2019).

[3] Marc G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *arXiv e-prints*, arXiv:1207.4708 (July 2012), arXiv:1207.4708. arXiv: `1207.4708 [cs.AI]`.

[4] Greg Brockman and John Schulman. *OpenAI Gym Beta*. 2016. URL: `https://openai.com/blog/openai-gym-beta/` (visited on 01/12/2019).

[5] John D. Cutnell and Kenneth W. Johnson. “Physics”. In: 4th. New York: Wiley, 1998, pp. 466–467.

[6] Haytham Fayek. *https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html*. (Visited on 03/04/2019).

[7] *GNU General Public License*. Free Software Foundation, June 29, 2007. URL: `http://www.gnu.org/licenses/gpl.html`.

[8] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-Learning*. 2016. URL: `https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389`.

[9] Sony Computer Entertainment Inc. *Cumulative Software Titles (as of June 30, 2008)*. 2008. URL: `https://web.archive.org/web/20080921062349/http://www.scei.co.jp/corporate/data/bizdatatitle_e.html` (visited on 01/12/2019).

[10] Junqi Jin et al. “Real-Time Bidding with Multi-Agent Reinforcement Learning in Display Advertising”. In: *arXiv e-prints*, arXiv:1802.09756 (Feb. 2018), arXiv:1802.09756. arXiv: `1802.09756 [stat.ML]`.

[11] Sergey Levine et al. “End-to-End Training of Deep Visuomotor Policies”. In: *arXiv e-prints*, arXiv:1504.00702 (Apr. 2015), arXiv:1504.00702. arXiv: `1504.00702 [cs.LG]`.

[12] Marlos C. Machado et al. “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents”. In: *arXiv e-prints*, arXiv:1709.06009 (Sept. 2017), arXiv:1709.06009. arXiv: `1709.06009 [cs.LG]`.

[13] Frank La Vigne (Microsoft). *Artificially Intelligent - A Closer Look at Reinforcement Learning*. 2018. (Visited on 01/12/2019).

[14] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1928–1937. URL: `http://proceedings.mlr.press/v48/mniha16.html`.

[15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (Feb. 2015), 529 EP -. URL: `https://doi.org/10.1038/nature14236`.

[16] OpenAI. *OpenAI Gym Environments*. URL: `https://gym.openai.com/envs/` (visited on 01/12/2019).

[17] OpenAI. *openai/atari-py*. GitHub repository for OpenAI’s adaptation of ALE. 2016.

[18] practicalcryptography.com. *http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/*. (Visited on 03/04/2019).

[19] Tom Schaul et al. “Prioritized Experience Replay”. In: *arXiv e-prints*, arXiv:1511.05952 (Nov. 2015), arXiv:1511.05952. arXiv: `1511.05952 [cs.LG]`.

[20] Joshua Walker. *Everything You Have Always Wanted to Know about the Playstation, But Were Afraid to Ask*. URL: `https://www.raphnet.net/electronique/psx_adaptor/Playstation.txt` (visited on 03/04/2019).

[21] Sen Wang, Daoyuan Jia, and Xinshuo Weng. “Deep Reinforcement Learning for Autonomous Driving”. In: *arXiv e-prints*, arXiv:1811.11329 (Nov. 2018), arXiv:1811.11329. arXiv: `1811.11329 [cs.CV]`.

[22] Ziyu Wang et al. “Dueling Network Architectures for Deep Reinforcement Learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1995–2003. URL: `http://proceedings.mlr.press/v48/wangf16.html`.

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

The Playstation Reinforcement Learning Environment

2384E, Jesus College

Originators: Petar Veličković and Cătălina Cangea

12 October 2018

Project Supervisor: Cătălina Cangea, Petar Veličković

Director of Studies: Prof C. Mascolo

Project Overseers: Prof A. Dawar & Prof S. Moore

Introduction

Reinforcement learning is the process by which an agent, given an environment in which to work, is able to 'learn' a policy that maximises the sum of its reward over the ordered set of actions that it takes. There are several real-world uses for the research in this field, but games have commonly been used as benchmarks to evaluate the effectiveness of certain algorithms. For several years now, a good source of test-cases has been the Atari-2600 console. Most notable among these uses is that of the February 2015 Nature paper [**1**] by a team of scientists from Google DeepMind in which existing RL algorithms were combined with deep neural networks, with the outcome being an agent that can demonstrate what they describe as 'better than human-level control' of classic Atari games. In recent years, research into RL algorithms has progressed to such a point that arcade-style games are no longer particularly challenging. Both the recent improvements made in RL (such as the Asynchronous Advantage Actor-Critic (A3C) algorithm in 2016 [**2**]) and the relative simplicity of Atari-2600 games provide a significant motivation to engineer and abstract newer and more complex environments in a similar fashion to that of the Atari-2600.

For this project, I propose that a suitable such environment is that of the Sony ® Playstation™ (PSX) console. There are a number of reasons for which this is a task of significant scale. Notably, the PSX is substantially more powerful than the Atari-2600 and capable of running more advanced games than those that have been used for RL purposes in the past. In practice, this means that most PSX games have far richer state representations than any games that run on the Atari-2600.

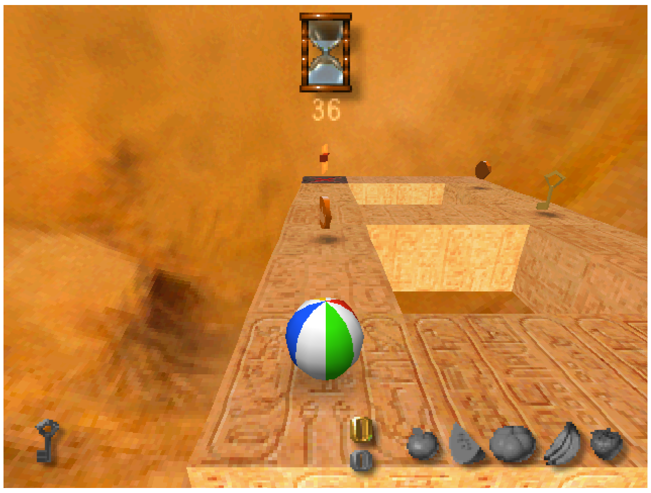


Figure A.1: The first level of Kula World

The game that I will be using with these algorithms is known as Kula World developed by Game Design Sweden AB and released in 1998 for the PSX. It primarily requires that the player navigate a 3D grid-like world with the aim of collecting coins, keys and fruit. Within the game, coins and fruit are worth ‘points’ to the player and some number of keys are required to proceed to further levels. This amounts to a game that has a clear reward structure, a simple action space and a clear imposed time limit. These factors make Kula World an ideal choice for an RL application.

In order to improve the accessibility of the Atari-2600 as a training environment for RL, OpenAI released the OpenAI Gym in 2016. This provides a relatively simple interface between a selection of Atari-2600 games and Python, allowing developers to interface (at a relatively high-level) with the gameplay. It would be of substantial utility to those in the RL research community to interface the full Gym API with my work; doing so would allow existing baseline algorithms as well as existing work using the Gym to be easily used in the PSX environment. The project involves several key stages of work. Firstly, I must work on adapting some existing emulator such that it can properly interface with the OpenAI Gym API. This involves making modifications that output the frame buffer’s contents; locating specific memory contents related to metrics such as score and position; and exposing methods through which controls can be affected within the game.

Secondly, I will look to extend the interface to fully support the OpenAI Gym API. This involves packaging the interfacing script in a suitable way, as required by the OpenAI specification.

Thirdly, I will work to deploy some algorithms. The focus will be on the ones included in the OpenAI Baselines, of a similar nature to *V. Mnih et al.* [**1**], which involve Deep Q-Networks for learning.

Substance and Structure of the Project

The project structure is as follows:

Core Objectives

The emulator modification stage of the project is perhaps the most pivotal to its success. I will be required to read through and understand a largely undocumented and already heavily modified emulator, with the intention of abstracting several key features of the emulator to a Python interface. These include:

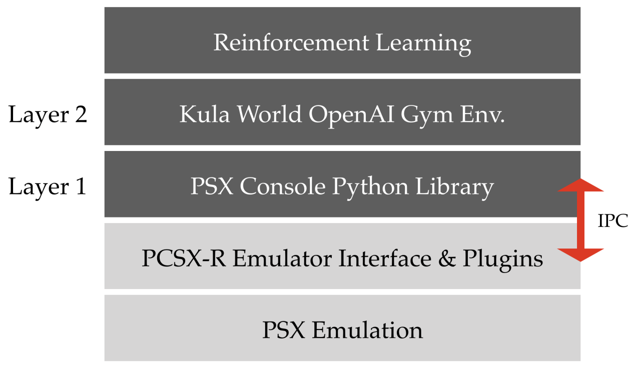

Reinforcement Learning
Layer 2
Kula World OpenAI Gym Env.
Layer 1
PSX Console Python Library
IPC
PCSX-R Emulator Interface & Plugins
PSX Emulation

Figure A.2: The layers of abstraction

- Exposing methods such that the Python interface is able to control the actions of the emulator.
- Studying the memory model of the PSX and collecting data so that I can work out whereabouts score information is held about the game.
- Extracting frame buffer information from the emulator so that I can both fully conform with the OpenAI specification, as well as pre-process frame data for use with a convolutional neural network later in the project.
- Investigate and implement IPC between the C code of the emulator and the Python. This should be done in a way that allows for multiple emulators to run simultaneously, though this is not required.
- Implement the OpenAI Gym API

Overall this project should produce three layers of abstraction to the emulator, as well as the IPC between the emulator and the Python library. In Figure 2, *layers 1 and 2* will be written in Python and the IPC will be written in both Python and C. Note that this approach allows for relatively simple modifications to be made to the *layer 2* such that games beyond Kula World can be used within this abstraction. This structure adds re-use potential to much of the code that I will implement.

I will also build an encoding of the first two levels of Kula World in such a way that it can serve as a spatial input to the neural network. Then, I can implement simple RL algorithms. I will implement a simple MLP neural network with `TensorFlow` and then use OpenAI Baselines to obtain gradients for learning.

It is possible that the emulator modification might be difficult, or that the structure of the emulator might make it too hard to retrieve the information that I require without substantial work. In such a situation, I will have to use fall-back measures in order to ensure the project can still be completed. These fallbacks are detailed below:

- Emulator not able to be modified for control : I can use 3D models of the worlds within the game and train the agent on a simulation of the game that I can build. Note that this would not be too much work, given the fact that the game is quite simple. During the testing stage, I can tabulate a set of actions that the agent would play and test these manually.
- Emulator not able to be modified for score extraction : Using Optical Character Recognition (OCR), I will be able to retrieve the score from the frame buffer of the running game. This can be verified in Figure 1 (above) where the score is visible (as a 0) at the centre towards the bottom of the screen. This will be relatively simple as many OCR libraries appear to be available for Python.

Extension Objectives

There are a number of extension objectives that I will try to complete should I finish the core objectives according to (or ahead of) schedule.

Firstly, I will attempt to implement more advanced learning approaches. Notably, I will use the frame buffer data that I extract from the emulator as input to a convolutional neural network rather than the hard-coded models discussed above. This will allow me to train the agent on more levels within the game which it would otherwise be difficult to hard-code maps for.

Secondly, I could attempt to use other sources of data within the game to act as percepts. For example, the PSX controller has DualShock support (a vibrating motor). This is used in Kula World to indicate to a player that, among other things, the floor below them is about to break. Also, sound effects are used to indicate the collection of coins and other rewards. Both of these features feasibly have the opportunity to improve the outcome of an agent that was trained using them. However, this may prove too difficult to implement due to the design of the emulator.

Starting Point

Primarily, the modification of the emulator relies on two part IB courses:

- Computer Design
- Programming in C

The Computer Design course is useful primarily in the capacity in which it discusses the mechanics of program execution. The Playstation console uses a MIPS R3051 CPU and so the interaction of the PSX registers with the MIPS Instruction Set Architecture (ISA) and with the built-in RAM have been covered in sufficient capacity to allow me to both understand and make modifications to the execution procedures.

Of course, the modifications that I make will not be to a real PSX device, but rather to an emulator. The emulator that I am likely to use is known as ”PCSX Reloaded” (PCSX-R). The emulation element of PCSX-R, while the executable is written using a variety of languages, is written overwhelmingly in C and shared across each platform’s build. This allows me an ideal opportunity to put the Programming in C course to use. The course focussed on features of

C such as pointer manipulation, data structures and memory allocation. Each of these will be very useful in aiding me with the required modification to the emulator.

Once I have sufficiently modified the emulator and successfully built a Python interface, I will begin building and running Reinforcement Learning algorithms that will interact with the game environment. This leverages two additional Tripos courses:

- Foundations of Data Science
- Artificial Intelligence

The Foundations of Data Science course made heavy use of Python; specifically the `numpy` library. This is a very useful library for manipulating matrices and working with datasets. Beyond the practicals in the course however, I have little experience using the `numpy` library and so I'll have to learn how to use it throughout the implementation stage of the project. This is specifically important as I will be extending its use-cases substantially compared to those in the course by employing it heavily for the execution of learning algorithms. When training the agent using Deep Q-Learning I will need to make use of Multi-Layer Perceptron Neural Networks (MLP Networks), a concept that was first introduced in the Artificial Intelligence course.

It is worth northing that RL background is not covered in any Tripos course. This will require me to spend additional time to familiarise myself with relevant algorithms and tools that I will be using for the project, these include:

- OpenAI Gym
- OpenAI Baselines
- Q-Learning
- Deep Q-Networks

Resources Required

My project will require the use of my own personal computer: a MacBook Pro (2017) with sufficient capacity to run the emulator; allow me to write and run both Python and C code; and connect to the Computer Lab GPU machines that will be provided to me by my supervisors for agent training. Mac OS 10.14 (Mohave) is installed on my laptop, but I will run a virtual machine of Ubuntu 18.04 LTS so that I can compile and run applications that are intended to run within the Linux environment of the GPU machines. I have a MacBook Air with a similar configuration available in the case where my main computer becomes unusable. I will manage all of the elements of my project that require writing code using `git`. Specifically, I will make use of a private `git` repository both for backup and version control. I will keep backups of both the code that I write as well as the dissertation in compressed and encrypted .tar.bz archives on my home server and on Google Drive; these will be uploaded regularly and kept until the end of the project.

Success Criteria

The success of the project should be judged according to the following criteria:

- By the end of the project I should have a working Python library that allows interfacing with the emulator (*layer 1*).
- The library should allow score/reward to be available upon each action and control over a virtual controller.
- A further layer of abstraction (*layer 2*) should exist and conform to the OpenAI Gym API I should be able to run (simply, without error) at least one simple RL algorithm on the environment that I have built.
- At least one simple RL algorithm (from *OpenAI Baselines*) should run and be able to perform actions sufficient to complete the first two levels (as described by the hard-coded models) reasonably[1].
- I should be able to implement at least one alternate approach to learning and this should *outperform* the satisfying algorithm from the above criterion.
- Each of the abstraction layers, as well as the learning algorithm should be able to support and exploit reading frame-buffer information from the emulator. That is to say, there should be at least one implementation of an RL algorithm that plays levels beyond 1 and 2 reasonably[1] while using the frame buffer (and *not* any hard-coded models) as input. Specifically, I will attempt to run the `deepq` baseline from *OpenAI* which gets particular use with the existing `cartpole` environment.
- Finally, I should be able to compare the success of differing RL approaches. This involves using more than one algorithm to train an agent and compare their eventual reward after a fixed number of training episodes. I will also compare the value of reward over training episodes between all algorithms, including against a random agent. This should allow me to draw conclusions about the RL approaches that I'm using as they apply to Kula World.

Timetable

Proposed start date is 18/10/2018.
Note: I plan to work on the full dissertation throughout the course of the project

1. **Michaelmas weeks 3–5 (18/10/2018–7/11/2018)** Investigate the emulator: locate use score in memory and work on fabricating controller input. Learn TensorFlow from online tutorials and locate relevant materials pertaining to reinforcement learning. Read subsequent papers to *V. Mnih et al.* [**1**] regarding current research in reinforcement learning in games.
   Deliverable: Modification to emulator that prints out score and can press buttons.

2. **Michaelmas weeks 6–8 (8/11/2018–28/11/2018)** Implement methods of facilitating appropriate IPC between Python interface and the emulator and build a Python library that allows for high-level interfacing with emulator features. Note that this is *layer 1* from Figure 2.
   Deliverable: *Layer 1* of abstraction in Python.

[1]This is hard to define, but the measurement that I will use will be that an agent works reasonably providing it can be trained such that it plays the level successfully (i.e. finishes with positive score)

3. **Michaelmas vacation** Implement the OpenAI Gym API using the first layer of abstraction. This should allow me to use the new environment (Kula World) within existing contexts that OpenAI Gym is used.
   Deliverable: *Layer 2* of abstraction in Python.

4. **Lent week 0 (10/1/2019–16/1/2019)** Prepare progress report.

5. **Lent weeks 1–2 (17/1/2018–30/1/2018)** Run OpenAI Baselines on the environment and learn how to specifically use TensorFlow as it applies to my project.

6. **Lent weeks 3–4 (31/1/2018–13/2/2018)** Implement a reinforcement learning approach for the environment that uses a MLP neural network built with TensorFlow.

7. **Lent weeks 5-8 (14/2/2018–13/3/2018)** Compare several RL algorithms within the context of the PSX environment. Investigate extension objectives (as above). Begin writing dissertation.

8. **End of Lent (14/3/2018)** Prepare an early version of my dissertation for feedback from supervisors and Director of Studies.
   Deliverable: Preliminary dissertation.

9. **Easter vacation:** Prepare and write a final copy of my dissertation for submission in Easter term. Send copies of near-final dissertation to supervisors and Director of Studies by the end of the vacation period.

10. **Easter weeks 0–2 (18/4/2019–8/5/2019):** Finish and submit dissertation.

Bibliography

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis. *Human-level control through deep reinforcement learning.* Macmillan Publishers Limited, 2015.

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning.* Cornell University Library, 2016.

Appendix B

Reference Specifications

ProcPipe Method Specification

ID	Name	Description	Supports Callback	Arguments
1	Exit	This exits the emulator while freeing memory from both the emulator and all plugins	No	N/A
2	Pause	Freeze the emulator's main thread	Yes	N/A
3	Resume	Resume the emulator's main thread from a paused state	Yes	N/A
11	GPU Snapshot	Take a GPU snapshot and copy it into GPU shared memory	No	Snapshot Console Discriminator Value (1 byte)
12	Clear GPU Snapshot	Clear the memory used for the snapshot	No	N/A
21	Memory Snapshot	Copy a section of PSX memory into shared memory	No	Start Index (4 bytes/int)* Length (4 bytes/int)* Memory Console Discriminator Value (1 byte)
22	Copy Memory to Disk	Copy a section of memory to a file on disk	Yes	Start Index (4 bytes/int)* Length (4 bytes/int)* n=Size of Path (1 byte) Path (n bytes)
23	Clear Memory Snapshot	Clear the shared memory used by the memory snapshot	No	N/A
24	Silence Memory Notification	Silence the notifications of the memory listener with a given ID; notifications will *no longer* be sent though NotificationPipe	No	Listener Key (1 byte)
25	Wake Memory Notification	Un-silence the notifications of the memory listener with a given ID; notifications will resume being sent though NotificationPipe	No	Listener Key (1 byte)
26	Write Byte	Writes a byte to PSX memory	Yes	Start Index (4 bytes/int)* Value (1 byte)
27	Drill Writes (unused)	Writes a byte to PSX memory repeatedly for a given number of 0.05s cycles	Yes	Start Index (4 bytes/int)* Value (1 byte) Number of Times (1 byte)
31	Start Audio Record	Begin audio recoding to a temp file	No	n=Size of Path (1 byte) Path (n bytes)
32	Stop Audio Record	Stop audio recording and close temp file	No	N/A
41	Save State	Save PSX state (registers, memory contents, PC) to disk	Yes	n=Size of Path (1 byte) Path (n bytes)
42	Load State	Load a specified PSX state from disk	Yes	n=Size of Path (1 byte) Path (n bytes)

* Note that for instructions, integers are sent in big-endian.

Figure B.1: This table gives the specification for packets on `ProcPipe`.

Game State

Value	Game State
48:	Playing
20:	Success
2 :	Spiked
6 :	Fallen off
4 :	Time up
18:	Game over
72:	Score Summary
50:	Main menu
52:	Pause menu
54:	Options menu

Figure B.2: This table describes the meaning of values of the *game state* variable.

CBPipe Response Specification

Value	Meaning
0	Memory notification follows
1	<RESERVED>
2	State load is complete
3	State save is complete
4	Memory successfully dumped to file
5	Byte written to memory
6	Drill complete
7	Emulator paused
8	Emulator resumed
9	Audio finished recording
10	Execution speed change complete
11	Memory read copied to shared memory
12	Aliveness check response

Figure B.3: This table gives the specification for packets on `CBPipe`.

Appendix C

Network Design

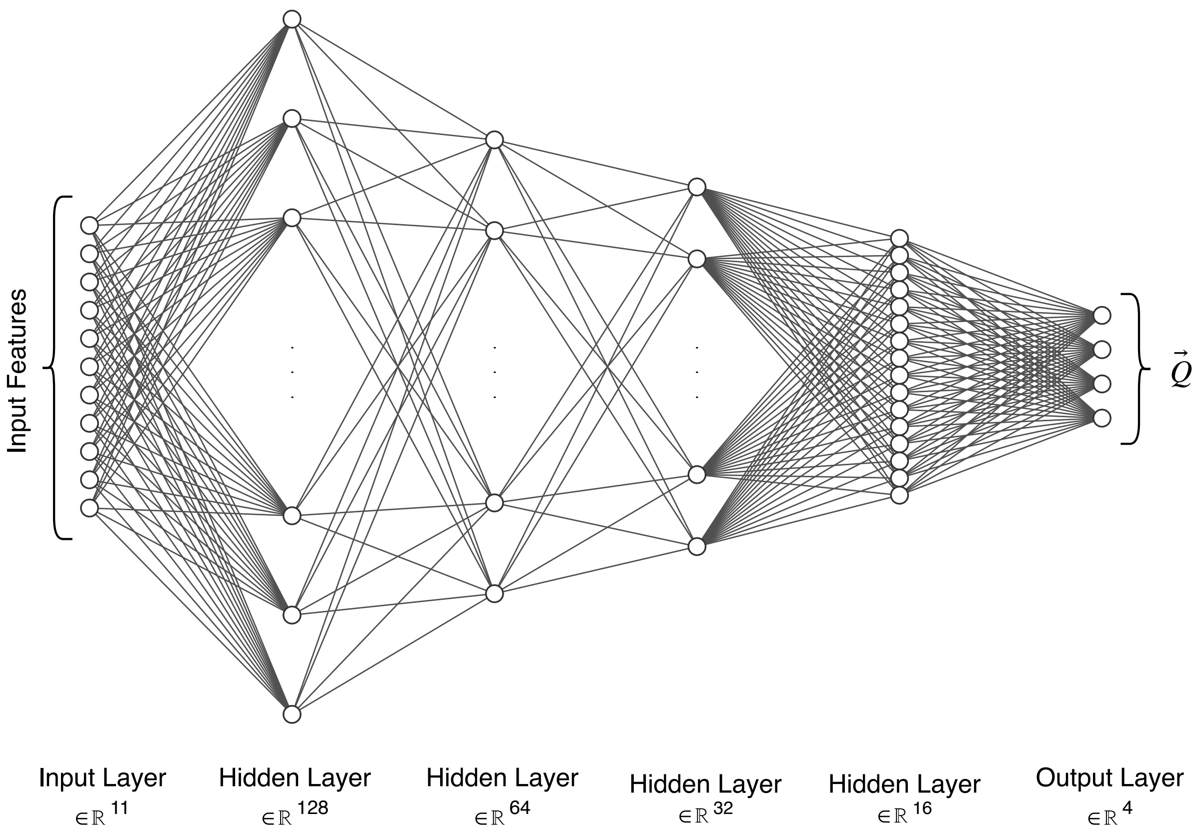


Figure C.1: This is the network used for the agent-local representation.

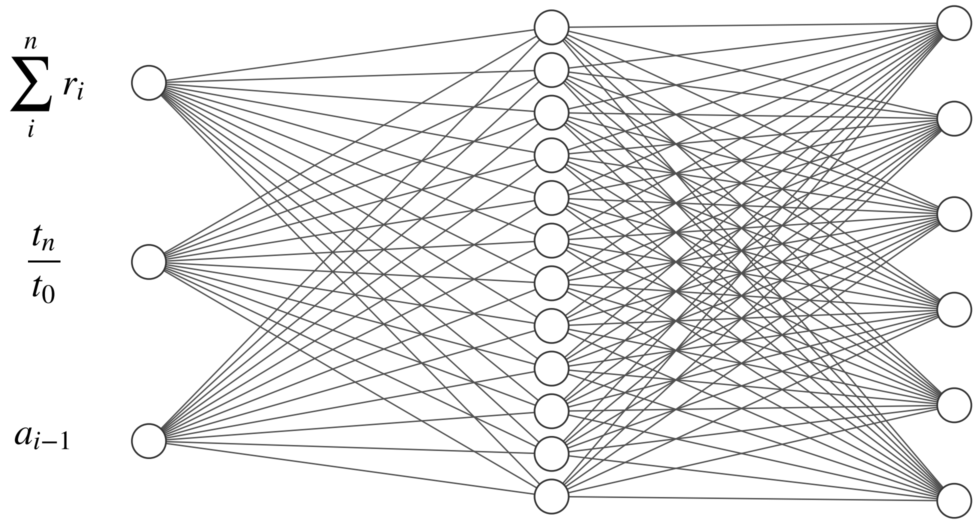


Figure C.2: This is the network used to interpret episode data. It has 3 inputs, 6 outputs and 12 hidden nodes.