**Your Project Title**



Harrison R Leitch

LEI16657607

16657607@students.lincoln.ac.uk

School of Computer Science

College of Science

University of Lincoln

Submitted in partial fulfilment of the requirements for the

Degree of BSc(Hons) Games Computing

*Supervisor:* Dr. Phil Carlisle

April 2022

**Acknowledgements**

Firstly, I want to thank somebody, and somebody else. Here is another thing.

**Abstract**

Here is the abstract for this project report.

**Table of Contents**

[Introduction 1](#_Toc102230189)

[1.1 Some notes 1](#_Toc102230190)

[1.2 Testing some mathematics 1](#_Toc102230191)

[1.3 Undergraduate Project Report 1](#_Toc102230192)

[1.4 Referencing 2](#_Toc102230193)

[1.4.1 Ludography 2](#_Toc102230194)

[Literature Review 3](#_Toc102230195)

[2.1 Background 3](#_Toc102230196)

[2.2 Related Literature 3](#_Toc102230197)

[Methodology 4](#_Toc102230198)

[3.1 Project Management 5](#_Toc102230199)

[3.2 Software Development 5](#_Toc102230200)

[3.3 Toolsets and Machine Environments 6](#_Toc102230201)

[3.4 Research Methods 6](#_Toc102230202)

[Design, Development and Evaluation 7](#_Toc102230203)

[4.1 Requirements and Game Design 7](#_Toc102230204)

[4.2 Unity ML Agents 8](#_Toc102230205)

[Difficulties with Unity ML Agents 8](#_Toc102230206)

[Conclusions 9](#_Toc102230207)

[Reflective Analysis 10](#_Toc102230208)

[References 11](#_Toc102230209)

**List of Figures**

3.1 A picture of the Brayford from Google Images. . . . . . . . . . . . . . 5

**List of Tables**

1. Here is a table. . . . . . . . . . . . 5

**Chapter 1**

# Introduction

Creating an agent which can work on all levels

## 1.1 Some notes

It is worth noting that this document is a project report template for the University of Lincoln, School of Computer Science. It should give you some direction and instruction for formatting and presenting your project report. If you have any suggestions or issues, please contact mdoughty@lincoln.ac.uk. It has been derived from the Latex PDF however, so there might be some issues – but ones I suspect you can overcome!

## 1.2 Testing some mathematics

Here are two equations using the equation editor (1, 2):

(1)

(2)

And here is some text with some nice inline maths, (*x,y*) wow *γ* so cool *ρ*.

## 1.3 Undergraduate Project Report

Currently, this template is set up for use with undergraduate project reports. However, the template can be modified fairly easily to conform to, for example, an MComp project report.

## 1.4 Referencing

It is worth noting that the standard for referencing is Harvard.

### 1.4.1 Ludography

There is an optional ludography for Games Computing students. To cite games, you can cite like any other reference with Harvard styling.

**Chapter 2**

# Literature Review

## 2.1 Background

Reinforcement Learning (RL) is a challenge which is being actively explored in relation to game agent creation. RL allows an agent to explore an action space to find the best combination of actions to maximize a reward function (Liu et al., 2021).

Deep Reinforcement Learning has been shown to successfully outperform a human expert in some Atari-2600 games (Mnih et al., 2013). Mnih et al. introduces Deep Q-learning which, in this example, takes an input of greyscale, down-sampled frames of Atari game footage, processing the last 4 frames through a convolutional neural network which estimates the best action to take (2013). Q-learning trains an agent using an experience replay which stores the state of the environment, the action taken by the agent and the reward gained at the next step (Mnih et al., 2013).

Proximal Policy Optimization (PPO) limits the change in policy per update and allows multiple epochs of update per sample as training occurs. This is implemented in more simply than its predecessor, Trust Region Policy Optimization, while also performing better in most tests (PPO).

Reinforcement learning can be improved by training an agent using a curriculum.

Camargo and Sáenz (2021) found that curriculum learning can both save up to 40% of RL training time and result in a better performing agent after training. Curriculum learning introduces the agent to behavior in a structured way, allowing the crafting of curriculums which can train an agent by presenting challenges in incremental difficulty. It was found that half of the 24 curriculums tested resulted in an agent which performed better than through training with PPO alone (Camargo and Sáenz, 2021). Patterns in curriculums that improve agent training included giving the agent an additional reward which decreases rapidly during training. This was done in the form of a reward for touching the ball. This encourages the agent to interact with the ball and then explore the rewards for pushing the ball towards the net once there was no reward to touching the ball.

Reinforcement learning has also been shown to be a useful testing tool. Reinforcement learning for load testing games (RELINE) is a recent development which trained an agent using a reward function with two objectives; advancing in the game and identifying states of the game which result in low FPS (Tufano et al., 2022). The system was tested using artificial performance bugs inserted into simple 2d games and real performance impacting bugs on an open-source 3D racer. It was found that RELINE allowed agents to effectively play and find performance issues in a game. The agent found all injected bugs in a version of *Ms. Pacman* in almost 90% of episodes (Tufano et al., 2022). RELINE also performed better in the racer, finding significantly more low-fps points than the base RL agent which had been trained in completing the game only (Tufano et al., 2022).

Imitation Learning (IL) techniques use demonstrations recorded by an expert to build a model that can act as the expert would. Lee et al. present an agent trained to play *Super Mario* using inverse reinforcement learning (IRL) (2014). IRL derives a reward function from the behavior of the demonstrator which is used to train the agent. It was found that the technique quickly converged, and the model produced matched state of the art human player imitation techniques in terms of being judged to be human (Lee et al., 2014). This technique, however, was storage intensive and trained on a simplified version of *Super Mario* therefore possibly making it inappropriate for training agents in more complex games.

Generative Adversarial Imitation Learning (GAIL) works in a similar way to Generative adversarial networks, in that a second agent is trained alongside the action taking agent which attempts to distinguish between “true data” and data generated (Ho and Ermon, 2016). In a similar fashion, GAIL rewards the acting agent for confusing the discriminating agent, resulting in an agent which behaves similarly to the expert demonstrations provided. This could be used in games to create functionality like *Drivatars* in the Forza Horizon and Forza Motorsport franchises in which a player can race against an offline player with a learned model of the players driving style (Playground Games, 2021).

A system can be trained thorough imitation learning without additional reinforcement learning to play *Super Smash Bros.* in a way competitive with the most difficult in-built computer-controlled player (Chen and Yi, 2017). A vision only model, using only the output to the screen as input data and keyboard input from an expert as the output. The game’s camera, and therefore input into the system, is complex allowing 3D rotations and zooms. The player and opponent characters were consistent and visually distinct (*Pikachu* and *Mario* respectively) which simplified the task for the network, the frames of gameplay video were also down sampled allowing a reduction in the size of the network. Each input consisted of 4 frames to allow the system to observe temporal information such as trajectory of movement. The system trained the agent quickly however the quality of agent produced and speed of training is proportional to the quality of the expert data provided (J. Harmer et al., 2018)

Imitation learning, specifically behaviour cloning, can suffer from “causal misidentification” (de Haan et al., 2019). Causal misidentification occurs when a model is not optimal due to the misidentification of the causes of expert actions (de Haan et al., 2019). This is often due to an overabundance of information input into a system in which the consequences of past actions are observable in the current state e.g. a driving model may learn to brake when the braking indicator is lit. This is an issue for behavioral cloning with a visual input (de Haan et al., 2019), such as in the research by Chen and Yi described earlier.

Imitation and Reinforcement learning have been used in combination to train an agent in a several games (Jacob et al., 2020). Harmer et al. combined these methods to train an agent in a first-person shooter game with a low-resolution visual input and an output capable of performing multiple actions at once (2018). This combination performed significantly better than pure IL and RL techniques as the agent can learn from it’s own experiences and is therefore not limited by the skill of the teacher, while also training faster than RL methods as the agent is given “prior knowledge about effective strategies” (J. Harmer et al., 2018).

The Unity Engine was chosen as the medium through which the game was built because of the extensive ML Agents package (Unity, 2022). This package has an implementation for reinforcement learning supported by imitation learning. The package provides implementations PPO and GAIL among other techniques. of The source code of all the models is available which allows the adjustment of models to test new machine learning techniques (Unity Technologies, 2022).The engine has also been used to implement the curriculum learning mentioned earlier (Camargo and Sáenz, 2021).

The Godot Engine was considered, however the Machine Learning package for the engine contained no imitation learning techniques (Beeching et al., 2021). The game *StarCraft II* has been used extensively in analyzing the performance of ML methods, an API has been created to expose the properties and state of play and allow ML models to train using the game (Vinyals et al., 2017). *StarCraft II* is a complex game, requiring mastery of micro and macro-management of troops and planning and building of a base. Replays consisting of state-action pairs from each step of the battles of highly skilled players can be used to train ML models (Justesen and Risi, 2017). The complexity of

The use of a side-scroller for the testing of machine learning techniques is well established, the *Mario AI Championship* used a version of *Infinite Mario* as a test environment for competing AI agents. The Championship tested AI in different “tracks” including “Gameplay”; how far an agent could traverse, and “The Turing Test”; to identify the most convincing human-like agents (Togelius et al., 2013). A benchmark and API for *Infinite Mario* which allows AI to be compared and tested (Karakovskiy and Togelius, 2012).

**Chapter 3**

# Methodology

Here is a sentence, and you can see a nice picture in Figure 1.



Figure 1: A picture of the Brayford from Google Images.

A table showing some data is displayed here (Table 1). This doesn’t have to be a table, it could be in the form of a chart or some other form of data representation.

|  |  |  |
| --- | --- | --- |
| **First name** | **Last name** | **Age** |
| Bob | Bobbington | 24 |
| Beth | Wavies | 49 |
| Joe | Bloggs | 37 |
| Billy | Bob | 10 |

Table 1: Here is a table.

This section will cover a number of aspects of your project where appropriate. **Not all projects will require every section though**. The key thing is that you demonstrate critical awareness of all of the processes that you have employed in your work and that for all sections needed in your report you are presenting a justification for the methods you adopted and not just presenting a list of methods.

## 3.1 Software Development

Managing the development of this project required the application of techniques from the domains of game development and machine learning. The project followed an agile framework which took inspiration from techniques specialised for the requirements of both machine learning and game development allowing the dynamic reallocation and scheduling of tasks and the re-evaluation of the requirements for tasks (Sommerville, 2011). The creation of the playable game was split into two distinct phases. The first followed the recommendations of the production phase from Fullerton, create a version of the game playable by a human, with a seed for the level chosen at build time (2018). The next phase of development was the integration of the Unity ML Agents package into the game to interface. This required modification of several of the methods and classes created in the earlier phase. In order to minimise the time lost through implementing the package, the example environments and documentation was studied alongside the first phase of development (Unity Technologies, 2021). This allowed alterations in the architecture of, for example, the player character implementation so that an Agent class could later be created to be made during or before the implementation was complete. The workflow presented by Amershi et.al. was followed in the later phase of game development as the Unity ML package was integrated into the game (2019, 291). This guided the crafting of the reward function and observations around the player character and the experimentation around these.

The planning and development of the game on which the agent is trained followed the stages of development set out by Fullerton (2018). This framework describes the game development process in a series of agile phases. These phases consist of the activities and areas of development which should be considered in this section. These areas should not be reconsidered or altered in a signific way after the phase is considered complete as this would require a significant alteration to both the elements themselves and the code and assets which are dependant. This approach was supported by the Gannt chart, whose early milestones represented the completion of the concept, preproduction, and production phases. As the game was intended to be used to investigate IL in Unity ML, phases past this were not considered. The specification and planning of the game occurred without issue, as the game needed to be a simple abstraction of a side-scroller. The production phase of the game which followed this framework progressed well and benefited from the guidance that the scope of changes considered to the game in development should be inversely proportional to the stage of development. As mentioned previously, the Unity ML package was studied during this time. It was found that the example projects created were significantly simplified examples of single behaviours e.g. jumping over a wall using a block. This led to a reconsideration of the difficulty of the levels implemented and the decision to not implement enemies to the game, in early development at least. The game’s structure was not changed, and other classes were implemented such that enemies could be added to the game at a later date with little difficulty. The omission of features is an acceptable change in scope at this stage of development according to Fullerton as the developers have a greater idea of what is possible within the time constraints (2018).

Machine learning comes software engineering implications not found in the production of other forms of software domains. These are presented by Amershi et al. as; The discovery, versioning, and management of data, the customisation of models to the current problem area, and the management of module boundaries between models (2019, 291). The authors present a workflow created from the practices of Microsoft teams specialising in machine learning. This workflow was used in the second phase of development of the game, while integrating Unity ML agents with the working game which had been created. The workflow has a distinct rooting in the agile framework, allowing feedback loops across model to allow for the experimentation inherent in machine learning to converge to a good model (Amershi et al., 2019, 291). The workflow allows one to loop back to any stage from the model evaluation stage, reflecting the importance of experimentation in ML model crafting. This was particularly taken advantage of in the experimentation later in the project as the models were analysed by, among other processes, visual inspection. This allowed the diagnosis of issues and unexpected behaviour and allowed the modification of the reward function or observations taken, which is discussed in 4. A significant portion of time was allocated to this period in the project, to increase the chance of a successful model as troubleshooting of neural network based models makes them difficult to debug in traditional ways (Arpteg et al., 2018, 50). Due to the nature of RL several of the stages of this workflow are achieved together or not needed. The collection of data, for example, was not a concern until IL were introduced, the Unity ML package makes the collection of demonstrations a simple task requiring a developer to simply play as the agent they are training. A greater concern related to this was the selection of observations, which takes the place of input data and is discussed at length in 4. Over the course of experimentation over 60 tests were run each testing a combination of hyperparameters, reward function, and observations.

* Early phase of dev was game and learning Unity ml
* Later phase was integrating ml into the game and crafting the reward function around this
* Sort of waterfall but with agile elements.
* Gannt chart helped visualise and laid out a rough plan
* Due to the nature of the project this was understood to change
* Agile plan changed, after reconsideration enemies should be implemented later in the project
* Agile. Changed mind after realising (small) complexity of unity examples

Github co-pilot

## 3.2 Project Management

Some awareness of project management should be demonstrated in all projects. This section should outline the nature of your project and the specific characteristics that need to be considered in determining what project management methodology you should use. You should identify the specific demands of your project in terms of project management and support your rationale for the selection of a methodology with appropriate and recent academic references. Questions which may be relevant here are:

1. What are the guiding principles and processes in managing your project?
2. What project management methods may be useful for this project?
3. How can you exploit their advantages for your project and mitigate their drawbacks?

* Version control
* Trello to keep track of tasks
* Regular meetings – very helpful
* Goals set, goals achieved each week
* Time left at the end for extensive development of model
* Machine learning takes time for issues to be ironed out
* Notebook for recording observations and planning

To aid in the execution of the workflows mentioned earlier, a Kanban board was used. This, along with regular meetings with the supervisor, allowed tasks to be added as the project needed them. These tasks were typically decompositions of the tasks laid out in the Gannt chart presented in FIGURE. Meetings were structured like scrums, typically ending with an agreement as to the next weeks work (Sommerville, 2011). The tasks agreed would be added to the Kanban board for completion or progress check by the next meeting. This became more difficult as the project entered the experimentation phase of development, as the actions taken to progress the project became more obscure (Arpteg et al., 2018, 50). At this stage the meetings became a discussion of the current state of the model and exploring theories as to the reason the model was failing. These theories would then be explored in the next week, with modifications to the reward function, hyperparameters, and observations.

The Gannt chart presented earlier provided a rough estimate as to the time each stage of the project would take as well as an outline as to the tasks needed to achieve these goals. The tasks presented in the Gannt chart followed the SMART framework. Before a task was started, it was discussed in a meeting, where it’s SMART criteria were evaluated and changed in order to better suit the project.

Version control was used throughout the project thorough Git Kraken. Git Kraken provided a GUI for navigating the version control of the project while also storing the project remotely. This allowed the project to be downloaded onto machines in communal spaces, allowing more than one model test to be run at a time. Version control assisted in the experimentation in crafting a model as it allowed the easy storage of previous versions of all modifiable parameters. Through this, a previous version of, for example, the reward function could be accessed and reintroduced if modifications did not have the desired effect.

## 3.3 Toolsets and Machine Environments

Toolsets refer to both software development and to project management, so the coverage should address both. This section will outline the tools for software development and project management process; it will make appropriate comparisons between tools available and argue for the most appropriate selection based on metrics, possibly a matrix diagram and other criteria. DO NOT justify the grounds for using specific toolsets and environments simply because you know them well or have developed skills already.

Also detail type of level training. Mario ai gym and open ai gym

The toolset used was largely dictated by the Unity ML Agents package. In choosing the machine learning package in which to root the project, alternatives were considered from the Godot and Unreal game engines. The Godot engine, as mentioned earlier, lacked IL functionality despite presenting more than 20 RL algorithms (Beeching et al., 2021). An Unreal Engine implementation of RL *MindMaker* also lacked IL functionality and has not been used in academic research before (Krumins, 2019). The Mario AI benchmark was created to provide a standardised testing environment for agents to complete a modified version of *Infinite Mario Bros*, a procedurally generated version of *Super Mario Bros* which creates levels and worlds (Karakovskiy and Togelius, 2012, 55). This environment has been used in competitions where typical AI approaches, such as A\* algorithms and Reinforcement Learning techniques have been compared (Togelius et al., 2013, 89). The benchmark generates complex levels, with multiple type of enemy and complex level structure, such as the dead end shown in IMAGE below. The API uses a grid system to input information to the models about the area around the player, IMAGE. Unity ML Agents can receive input from several sources, including ray casts and collisions as well as allowing inputs to be given sporadically. This greater flexibility in the available inputs as well as the ability to implement a Mario-like side-scroller with control over all aspects of the game design and implementation through the Unity Engine made the package. The Mario AI Benchmark was used to inform the development of the game and is discussed further in SECTION. Unity ML Agents required the use of a python environment with the installation of Pytorch and TensorBoard for the creation of models and the viewing of information regarding the training of the agents respectively (REFERENCE?). A critique of the Unity ML Agents package is presented later.

Trello was used to manage the Kanban board, described earlier, used throughout the project. Trello is an in-browser application which allows easy access to the information at short notice from any device (Johnson, 2017, 209). As this was an individual project, the shortcomings associated with assigning tasks to team members was not present

Toolset decided by unity ml agents

Everything is included

Godot, unreal alternatives

Explain python and .yaml and cmd line interfaces

Trello and git kraken for project management

Include why referecnces have been referenced, and include images from papers – even with facts

Next steps in project, Improve unity ai? Explainability? Better more challenging environments, step size and training speed.

Discuss RL in games in general where is the industry? Put early and a further discussion later. In intro mention explainability then a further discussion later. Not original intention, project shifted to focus on this through experimentation

Black and white- early perceptrons in games

BC then RL test

## 3.4 Research Methods

Pretty certain this section should be more broad than this, too into the fine points. Also detail experimental setup. Other Benchmarks. Maybe move the exact metrics?

In the analysis of the models produced, several metrics were used. The agents consistency in completing a level was considered as this is the first indication that a successful model of a solution to the problem has been produced. It is the first metric considered for determining a winner in the Mario AI Competition (Togelius et al., 2013, 89). The next parameter considered is the total number of jump events compared to the expected number of jump events. This can aid in the determining of whether an agent which completes a level is behaving as expected, rather than jumping continuously. This metric closer analyses the success of IL and its similarity to the demonstrations presented. The training of the model was also evaluated. The average reward gained during training provides several insights as to the quality of the model. While the value of reward cannot be compared across different reward functions, observing the reward earned as training goes on allows us to see when a model converges close to its final values. This value is naturally noisy as the model trains because reinforcement learning uses a curiosity function to explore the problem area so, in support, the estimated action-value function or estimated extrinsic reward can be used (Mnih et al., 2013).

These metrics will be used alongside observing the model complete levels after it has trained to diagnose issues with the behaviour of the agent. As the game is the domain in which the agent is prepared for, the observation of the agent as it completes the level is equivalent to observing a self-driving car. It also allows comparison against the demonstrations presented when IL is introduced. Lee, et al. use a survey to garner opinion on whether the model is human-like, however as the aim of this project is to create a model using IL, a simple observation and comparison with the expected number of jumps will be used to measure similarity to the demonstration data (2014, 1).

In the analysis of the training, the time taken to settle on a value

The value of the reward signal can be used for like for like reward function comparison

Though this should be paired with visual observations

Record avg. expected jump events against expected

Complete vs fail level

You should investigate the types of research methods necessary to validly answer the research questions that your project addresses. You should cite relevant sources to justify your choices.

**Chapter 4**

# Design, Development and Evaluation

This section of the report will vary significantly in both structure and content, depending on the type of project you are undertaking. For example, a Games design project may include a Game Design Document. However, it must be noted that if your project contains significant software development work, this should be presented in the structure expected of a formal development report. If your project involves an experimental evaluation – especially if that evaluation involved human participants – you are expected to write this work up in the format expected in Section 4.2.

## 4.1 Requirements

Requirement of game and project as a whole

It has been established by several other researchers in this area that to successfully train a model, a game must be a simple as possible. Lee et al. (2014, 1) use a simplified version of *Super Mario*, inputting a 5x5 grid of blocks around Mario while fixing his state to “big”, Chen and Yi (2017) use the same game stage of *Super Smash Bros* with the same characters and a down sampled image input of each frame. This informed the design of the game as a simple 2D side-scroller, based on the Mario AI Benchmark (Karakovskiy and Togelius, 2012, 55). A hand-written design document detailed the features of the game to be created. This included specifications of the level design, enemies, and the actions available to the player. Where other ML techniques require significant amounts of labelled data to effectively model their problem domain, RL uses trial-and-error to explore and optimize it’s model to maximise the reward function earned (Torrado et al., 2018, 1).

An agent which can perform as a human player across levels needs to be trained on a wide variety of levels. Procedural content generation (PCG) can be used to present the agent with an effectively endless variations of levels which will allow the agent to generalise its behaviour (Risi and Togelius, 2020, 428). Procedural generation is a common technique used in games to extend the amount of content present in a game, while reducing the amount of work needed in areas such as level design and asset creation (van der Linden et al., 2014, 78). *Enter the Gungeon* (Dodge Roll, 2016) is a bullet-hell dungeon crawler built in the Unity Engine which uses procedural generation to present a player with a consistently different series of rooms and enemy placements. The Mario AI Benchmark also uses procedural generation, as mentioned previously, to generate both levels and worlds. It was established that procedural generation in the side-scroller would be through a deterministic method, using a seed to allow repeatability and consistency across tests providing an accurate reflection of the quality of model and training parameters. The *Mario AI Championship* scored entries on a series of levels from the same seed for these reasons (Karakovskiy and Togelius, 2012, 55). The implementation of this system was also decided to be easily adjustable for the modification of difficulty, to allow a steady development of an agent rather than attempting to create a fully functional general solution immediately.

Causal misidentification can be a significant problem when dealing with IL and RL as de Haan et al. demonstrate, in even simple benchmark games such as *Pong*, that increasing the amount of information available to the model leads to inferior performance (2019). The lack of quality of performance comes from incorrect associations this can present as simple misunderstanding as to the observation which caused the action or, through observations, models can learn to execute actions from stimuli that occurs as a result of those actions, e.g. a driving agent could learn to brake only when the brake indicator light is lit if using visual observations as input. The variables observed by the agents was not decided during the planning and development stages of the game. Therefore, to allow experimentation with observations and to reduce sources of causal confusion through presenting the agent with as little excess information as possible, art was kept simple with no background and a basic sprite set.

Player character requirements

Move me

Levels are procedurally generated using a seed, allowing the same series of levels to be used by multiple training instances. A level consists of a series of tiles, each containing a feature or combination of features described by Dahlskog and Togelius (2012). These tiles are categorised by jump event and the expected number of jump events needed for the player to successfully complete the tile. For example, a tile could require 3 Double Jumps or 1 dash jump to navigate. This can be used to assess the agent’s behaviour throughout the level and will be discussed later in SECTION. The game was designed with focus on easy difficulty adjustment. The process to change the possible tiles in a level is a simple procedure allowing the removal and implementation of level features, such as bottomless pits. Structuring levels in this way made implementing the Unity ML-Agents package simpler. A level of consisting of a single repeating tile allowed the implementation of the interface with the package to be tested more rapidly than a complex level as an agent would take less time to train to a proficient standard.

Include this section if you are undertaking a software development project. You should discuss:

1. Requirements elicitation, gathering, collection and analysis
2. Design
3. Building and programming
4. Testing
5. Operation

## 4.2 Game Design

Levels are generated in a grid-based method. To produce mechanics and levels that are interesting, varied, and representative of side-scrollers, in particular *Super Mario* the patterns described by Dahlskog and Togelius TABLE (2012). These patterns describe frequent mechanics in the level design, such as stairs, valleys and gaps, which when implemented together present challenges for the player to solve. To introduce these patterns to the game, levels were generated from a tile map of 6x6 chunks. Each of these chunks contained a pattern described by Dahlskog and Togelius. An example level and the tile available for use in the procedural generation of levels can be seen below. The spawning of chunks followed a small set of rules:

1. A level began and ended with a completely flat tile.
2. A level could only be 3 chunks tall, this prevented a level from becoming too large and encroaching on other training environments present, which is detailed further later.
3. Levels had a set length which could be altered in the Unity editor. This was to prevent the creation of a level which would take a player too long to complete.

These rules manged the levels, ensuring the player would not be spawn killed with a gap underneath their spawn or reach a jump which it were not possible to scale. Tiles were categorised in the Unity Editor through the type of jump needed to complete them, the number of jumps expected to scale this obstacle, and whether it should result in the next chunk being spawned above or below the current chunk. These categories are shown below. This method allows simple classification of new chunks and clearing of a certain type of obstacle to view the agent’s behaviour. This was exploited in the early testing of RL models to produce a single repeating level to reduce the complexity of the model required to complete the level allowing a gradual build to a general model which could be used for any level. This is explored in detail later.

The player character implementation was created

More detail about Player character integration, jump types, parameterizable speed. The ML agents dictated this class be a little open

## 4.2 Unity ML Agents Integration

Implementation

The implementation of the Unity ML package into the game required little modification of the player character described earlier, instead adding components to increase the functionality of the object. The “Agent” base component is the main interface between the Unity ML package and a game. The programmer creates a component which inherits from the “Agent”, shown in FIGURE(use a UML diagram of the inheritance of agent and SideScrollingAgent), and overrides a series of functions to allow the model to interface with the character in training and the release of the game. These functions have a significant impact on training, defining the reward function and observations of the agent. This means that during the development of a project these functions are in a state of constant flux as the programmer experiments. The functions integral to the training of an agent are detailed below, along with a description of their original implementation in the side-scrolling game.

### Agent.CollectObservations(VectorSensor sensor)

This function is called earliest during the update cycle, providing the model with observations of the current state of the environment. The programmer can give the model information held in numerical values. Observations made using a component which implements the ISensor interface or through the *[Observable]* flag do not need to be included in this function as the package handles their input by default. The first implementation of this function can be seen below IMAGE, and includes observations of the agent’s current speed, position and jumping state. This function allows observations to be made inconsistently, as the VectorSensor modified within the function uses 0 values for each observation by default. While this feature is never used in the side-scroller implementation, the observations of the jumping state of the player character could have exploited this functionality by only inputting a value when the calling of a jump event is available to the agent.

### Agent.OnActionRecieved(ActionBuffers actions)

This function has three purposes; executing the action from the inputs of either the player or model, defining the reward function, and the calling of the method to end the episode. The model interfaces with this class through the ActionBuffers object passed in. This contains an array of numerical values, which are the outputs of the model and are used to determine if an action should be called through a simple logic check such as seen in the implementation shown below IMAGE. The ActionBuffers object contains continuous and discrete values outputted by the model, both were used in the control of the agent. A continuous value is used for the horizontal movement of the player, allowing the agent to move with a variable speed and in positive and negative directions. Discrete values are used to instruct the agent in jumping, these provide binary input through a logic check presenting the model with the same “button” as a human player.

The reward function is also defined in this method. The reward function first implemented was very simple, rewarding a player for finishing the level, punishing the player for falling off the level and giving the player a constant small negative reward for being in the level. The punishment given to the player over its time in the level was designed to encourage the agent to complete the level as quickly as possible. This simple reward function was not successful, and a full account of its development from this state can be found later. A reward function often requires significant tuning by the programmer before a RL based agent is optimised. Recently, techniques have been investigated to shape the reward function through meta-learning, these are beyond the scope of the project however could be considered a future direction of development of this project considering the success of early tests by Zou et. al. (2019).

The function is also responsible for calling the EndEpisode() function, which leads to the execution of all the methods needed to inform the model of a completed episode and the BeginEpisode() function which resets the environment. EndEpisode() is called when the episode can be considered completed before the maximum number of timesteps has been executed. In the original implementation of the OnActionRecieved() function end episode is called after the agent has less than 2% of the level left to complete (around halfway through the final chunk) or when the agent has fallen through the level. The package also automatically ends the episode when an established number of update steps have been taken.

### Observations

As mentioned above, Unity ML allows observations through components which implement the ISensor interface. The agent uses a RayPerceptionSensor to observe its surroundings. Ray casts are broadcast around the player and the information they return is used as input in the model. This is a cheaper way of observing the environment than a CameraSensor component, which uses images from the active camera as the observations closer replicating the deep reinforcement learning done by Minh et. al. (2013) or the Super Smash Bros agent trained from gameplay frames and keyboard input by Chen and Yi (2017). Visual input is also more susceptible to causal misidentification from the superfluous information stored in the image. The documentation for the Unity ML package recommends using as few rays as possible. 13 rays are cast for a distance roughly equal to that of the screen for the human players.

### Training Environment Instances

To increase training speed, multiple instances of the game can be run at the same time. A prefab of the level is used, ensuring consistency of the training instances. The documentation for the package recommends a small number of environments running simultaneously to improve the real time performance of training as several experiences are gathered in parallel. To balance computational load and memory requirements with the increased speed, 6 training environments were used simultaneously in the majority of tests completed. These environments were active in the same game instance so had to be placed at a far enough distance apart such that they would not interfere with each other.

### Demonstrations

The Demonstration Recorder component which can be implemented alongside the overridden Agent component. This allows the recording of demonstrations from within the Unity Editor allowing a designer to easily record actions to train through IL. The designer can record for a specified number of steps or indefinitely, saving the demonstrations to a .demo file in the specified directory.

### Hyperparameters

IMAGE, shows the original set of hyperparameters used in the training of an RL agent. Due to the number of hyperparameters available for experimentation, the first set of hyperparamters used were a copy of one of the sets used in the example problems of the Unity ML package. This was taken from a problem with similar observations to the side scroller tested to give greater relevance to the hyperparameters chosen. These hyperparameters were modified during the experimentation and a new set of hyperparameters was chosen from the example environments whenever a new type of learning was first started.

Testing

An agent observes several parameters about the state of play in the level. The agents speed, current jumping state, and a series of ray casts around the player character. These ray casts return if they collide with the level tiles visible to the player. These observations are stacked to give the agent temporal perception (REFERENCE). These observations expose all necessary information to the agent while obscuring unimportant information which could lead to causal confusion (de Haan et al., 2019). Using a series of parameters as inputs was chosen over a deep learning method of using the visual output from the game due to the additional complexity required to process this input and create a proficient agent. Deep learning methods are more prone to causal misidentification (de Haan et al., 2019) and require a significant increase in the compute time of training ().

The Unity ML Agents package provides a base class for agents through which an agent can be controlled while training. The class provides several functions to override through which one can input observations, set a reward function and use the output of the model as an input for actions.

Agent.CollectObeservations (VectorSensor sensor)

This function allows the addition of observations that are not collected through ray casts or other external means as these are identified for collection within the unity editor. The agent’s current velocity along both axis are used as input, along with the current jumping state of the agent. During the development of the game, the agent’s position along the level was observed as a parameter. This was removed as the position of the agent relative to the start of the level should not affect the action an agent takes and the information therefore increases the likelihood of causal misidentification. This resulted in a more unstable training performance which also did not converge on a solution. Test 29 and 30 were taken with these observations and their reward as training was carried out is shown in FIGURE. This could be because without this observation, the agent is not rewarded enough for traversing the level and so becomes stuck at local maxima of reward. This cannot be definitively proven, though we can observe the behaviour of the agent from this training by inputting the resultant model into the agent model parameter within the interface provided in the Unity Editior (CHECK THIS AND PROVIDE AT LEAST A SCREEN SHOT OF HOW TO ADD BEHAVIOURS). When a reward was implemented at the end of an episode for the distance travelled across the episode the training of the agents was expected to be significantly more successful, test 31 to ?? tested different versions of this reward function.

Agent.OnActionReceived (ActionBuffers)

This function deals with the actions that a model outputs and assigns reward to the agent in response to the state of the environment. The model can input continuous and discrete actions, both were used in the control of the agent. A continuous value is used for the horizontal movement of the player, allowing the agent to move with a variable speed and in positive and negative directions. Discrete actions are used to instruct the agent in jumping, these provide binary-like input providing the agent with the same input limits as a human player.

The reward given to an agent is also defined within this function. The reward function is significantly different from the first iteration as it is a key part in successful reinforcement learning. As mentioned previously, the reward function had to be adjusted to accommodate for the removal of the observation of the players location within the level. This caused subsequent training attempts to fail, without finding stability or a successful path to the end of the level. On every modification of the reward function, a new model was trained with reinforcement learning to both ensure that an agent can learn a successful or at least increasingly successful strategy to complete the level as well as providing a baseline for comparison with any model with imitation pre-learning completed thereafter.

The first major issue fixed in the reward function was the tendency to train an agent to repeatedly jump behind the start of the stage, ending the episode almost immediately. In response to this, a negative reward was introduced for falling off the stage. The negative reward was only given when an agent fell at a point before the start of the level, if the agent falls after the start a significantly smaller negative reward is given to encourage the agent to keep exploring around this action space.

An attempt at implementing a reward function that does not directly reward for the position of the player used a technique inspired by the reward shaping of the agent to find performance loss in (THAT PAPER), in which the agent is rewarded for finding each point of lag only once. The agent was rewarded every x time steps if its position was greater than it had been at any multiple of x timesteps. This reward was proportional to the distance travelled across the level. The number of steps between analysis was parameterised and could be set within the prefab of the level instance. The intended consequence of this component of the reward was to incentivise the agent to explore in the positive direction while de-coupling the reward to simply the distance travelled. In this state the agent can still be rewarded if it needs to move backwards to move forwards, for example if the agent misses a jump. Test 35 trained reinforcement learning only agent on a simple level. In this Test, the number of steps between a reward for distance was 50. FIGURE shows the reward given to this agent as it trained. The training was unsuccessful and unstable, producing an agent which could not navigate the level. In the next test, the reward given for the distance travelled by the agent was reduced. This produced an agent which successfully learned to navigate the level. This affirmed that the reward function can be used in reinforcement learning to train an agent to complete a simple level.

An imitation agent was trained to complete a single level using demonstrations of a human player completing the level. This also produced a successful agent and was verified over several models. The imitation model for this appeared to converge on a solution in a more stable manner than the reinforcement learning model, a comparison between the results of training of these methods is shown in FIGURE. The agents reach a mean reward close to that of their respective solution in a similar time, the curve of reward per episode plateauing at a similar number of steps. The agents display significantly different behaviour however, demonstrated in the difference between the average length of episodes. The reinforcement learning agent completes the level significantly quicker than the imitation learning model, taking around 800 steps per episode against the IL models 2.1k. Upon observation of the models produced, a significant difference between the behaviours generated was noted. The RL model moved rapidly through the level, using the dash and double jump frequently. The agent was observed to have learned some complex interactions in its ability to wall jump. This was an exploit which was not demonstrated to the IL agent which could allow the player a large vertical boost through resetting the Jumping State of the agent while it was colliding with some walls. As this agent was trained on the same scenario each episode, this behaviour could be seen as a form of overfitting. When this agent was trialled in a more complex level (LATER ON), the agent could not reach the end of the level and frequently fell of the stage after a wall jump. The IL agent also successfully completed the level when observed though behaved significantly differently to the Rl model. The agent moved through the level at a pace significantly slower than the RL agent and did not demonstrate that it had learned how and when to wall jump. The agent was also conservative with its use of the dash jump ability, using it to jump from one raised section to another. This is a behaviour which was displayed in the demonstration used to train this agent.

|  |
| --- |
| Graphical user interface  Description automatically generated |
| Figure 2:Mean Reward of Test 36 and 41 during training |
| Graphical user interface, chart  Description automatically generated |
| Figure 3:Average Length of Episode Across Training during Test 36 and 41 |

Agents were then trained on a more complex level. This level was generated using a seed with the complete tile set available for use. This allowed agents to demonstrate different behaviours depending on the tile presented to them and prove that an agent can learn to complete a single complex level. The level introduced challenges to the agents that were not present in the first level, including; a wall only scalable through a double jump, large pits which can be traversed using either a dash or double jump, and a time limit of 6000 physics updates which makes the speed of an agent more important. The simple RL model could not successfully converge on a solution which would consistently result in the agent reaching the end of the level. The mean reward gathered while training does not increase in a stable manner and tends to decrease slightly towards the end of the training session. As reinforcement learning relies on the curiosity system within the model to explore the problem space, it is expected that with enough compute time a solution could be found (Unity ML Agents comparison of techniques).

The IL model from Test 41 was given new demonstrations of a human player completing the more complex level. This agent progressed along the level in a steadily, successfully navigating obstacles however fails to reach the end of the level

Just a quick note: test 54 and 55 were of the new reward function without and with BC. They might not have worked as expected though you need to look at the models produced to determine this.

INTEGRATE INTO DISCUSSION OF COMPLEX LEVEL

The disparity between the speed of completion of the level is likely due to GAIL encouraging a resulting agent which mimics the demonstrations recorded rather than finding the optimal model for maximising the reward given in through the completion of the level in the reward function defined in OnActionRecived. To produce an agent which favours imitation of demonstration or maximising the explicitly defined reward function more, the hyperparameter Gail: strength can be adjusted. A higher GAIL strength will result in an agent which follows the demonstrations over an optimal solution.

## Experimentation and Tuning of ML Model

To develop a ML model to enable an agent to complete a general level, extensive experimentation with the hyperparameters, reward function and observations were undertaken. Testing was run using increasingly complex scenarios to aid the diagnosis of behavioural issues. Testing began on a RL only agent being trained on a level which consisted of the same chunk repeated. Testing on a complex level was then executed, this time with a pure RL agent and an IL and RL agent and, finally, a level which changed after each episode to create an agent which could generalise to solve any combination of level. This approach of gradually increasing the complexity of the problem on which a model is trained was informed by Tufano et. al. and the creation of an agent which could spot performance issues in games (2022). The authors first tested their model on simple games often used to test ML for games which had artificial performance issues injected for the agents to find. This allowed them to verify the validity of their method before testing on an open-source 3D racer without artificial issues injected.

### Initial Tests

The first set of tests to develop the model were run on a short simple level, consisting of a single repeating chunk. The original combination of hyperparameters, reward function and observations did not produce an agent which could successfully complete a level. Several hyperparameters, including the learning rate and number of epochs per batch of training data were changed with no impact on the quality of behaviour learned. While these tests were taking place the agent was observed to explore the level, moving further along then as the model progressed, began jumping backwards from the starting position, ending the episode immediately and being given a reward of -1. It was theorised the agent would learn to advance through the level to increase the value of reward earned, maximising this by slowly moving through the level before training collapsed. This suggested that the reward given for progressing through the level was too high. The reward function was modified to reduce the magnitude of the reward earned through the position of the agent in the level.

|  |
| --- |
|  |
| Figure 4: Reward Function Modification of Reward for Distance Travelled |

The model, again, failed to produce a successful agent. The model was observed during training to almost complete the level, occasionally passing the end of the level and falling off the end of the stage. This implied that the reward function could result in an agent which completes the level, so the training environment was investigated for issues and a bug was discovered which resulted in the incorrect setting of end zones. The end zones were being placed a significant distance from the final chunk of a level meaning an agent would have to execute a perfect leap in order to complete a level and get the reward. This was rectified and the test was run again, this time resulting in a successful agent. The Reward function and hyperparameters were therefore considered suitable for further experimentation in more complex environments.

|  |
| --- |
|  |
| Figure 5: Average Reward During Training for the Experimentation of Progress Reward |
|  |
|  |

the maximum number of steps per episode was increased from 3000 to 4000. This gave a maximum real time limit of 80 seconds per level as an action from the model was called every physics update. This resulted in a successful agent, seen in Figure 4. After the time limit had been extended the agent could explore the level for longer, increasing the chance of reaching the end and being rewarded. After the agent had consistently began reaching the end, around step 600000, training became an optimisation of the time taken to reduce the negative reward given per timestep spent in the level.

The peaks in reward around step 1x10­6 coincide with the longest episode lengths by the model limited to 3000 steps per episode. This suggests that, during training, the model begins to learn to move through the level to increase its reward however the constant negative reward from the agent not having finished the scene results in the disregarding of the expected behaviour. As the agent would simply jump backwards off the stage it could also be concluded that the agent prefers to end the episode as quickly as it can to prevent a lower reward from being earned. The negative reward was implemented under the guidance of the Unity ML documentation, stating that time sensitive tasks should be trained with a small negative reward each step. The documentation also states that an excessive negative reward can cause a failure to learn expected behaviours, further exemplifying the need for comprehensive experimentation.

### Complex Level RL Testing

The seed for the complex level was chosen such that the agent would be presented with only single and dash jumps, to avoid overloading the model with information which would make the diagnosis of issues harder. The RL model carried over from the simple level was not able successfully complete the level. The model experiences stages of successful learning before the model collapses, visible through the reward earned as the agent trained. The first peak coincides with a period of extended episode length, implying that, as the reward is less than 0, the agent learned to stay in place. The second peak reaches just above 0, and has an average episode length significantly less than its earlier peak. This implies that the agent was progressing through the level, gaining reward through the component of the reward function described in Figure 4, then falling at the second gap adding a negative reward of -1. After the agent fails to improve from this the model collapses, the agent being observed to jump backwards off the stage upon the beginning of an episode consistently earning a total reward of -1. This phenomenon had been observed in the testing of the simple level before the level end bug had been fixed, and therefore when the agent would constantly be punished for falling off a level. This suggests that, during training, the model begins to learn to move through the level to increase its reward, however, being faced with a seemingly unavoidable negative reward results in the disregarding of any reward increasing behaviour in favour of ending each episode as quickly as possible. This is supported by the Unity ML documentation which states that an excessive negative reward can cause a failure to learn expected behaviours (Unity Technologies, 2022).

|  |
| --- |
| Chart, line chart  Description automatically generated |
| Figure 6: Mean Reward earned During Training of the Simple RL Model |
| Chart, line chart  Description automatically generated |
| Figure 7: Episode Length During Training of the Simple RL Model |

Another consideration as to the cause of the agent failing was the size of information given to the model during training. The batch size and buffer size were modified simultaneously in the hyperparameters of the model. Increasing these values increased the number of experiences recorded before the model is updated. Batch size controls the number of experiences processed by the model during each update, while the buffer size controls the amount of experiences collected before the model is updated (DOCUMENTATION REF). There was very little effect on the quality of training through the altering of these hyperparameters. The largest of the set appeared to perform better, however did not learn to leap the first gap, instead waiting at the edge while the others experience the issue of leaping backwards off the level. It was therefore concluded their modification could not help the agent to converge on an optimal solution.

While the previously discussed tests did not provide a solution to fix the model, the agents tendency to stand still lead to the experimentation with the curiosity hyperparameters of the agent. This model supports PPO by providing a reward signal for ……. The higher the curiosity strength value, the more reward an agent is given for exploring the problem area thus encouraging new behaviours to be learned and avoiding local maxima and minima of reward. However, an overly large curiosity value can be detrimental, overwhelming the reward from the environment defined reward function and causing no behaviour to be learned (UNITY ML PAPER). It was found that the modification of the curiosity strength did not improve quality of agent produced.

Test 15 through 18 for playing with the curiosity value, this had no effect so the reward function was changed to remove punishment for falling in a hole. This stopped the suiciding but gave strange behaviours. This meant BC was started. No improvement so reward function and obs were reworked around test

Early model, starting with a simple level and simple RL model, how did the Reward function develop?

Simple level Imitation learning

Complex level RL then IL model. Describe why GAIL and BC were used when introduced

Just before final model, I think the IL failed because the agent never learned that the end of the level was good, so it maximised what it could get by moving slowly but constantly.

## 4.4 Agent

## 4.5 Analysis of Agent and Final ML Methods

## 4.6 Critique of Unity ML Agents

Problems with unity ML agents

The Unity ML Agents package provides a general interface for applying machine learning techniques to teach behaviour to agents created within the Unity Engine (Unity, 2021(ML AGENTS)). The functionality of the package is divided into 3 interfaces: the unity editor, c# classes which can be overwritten, and a command line interface. The package uses PyTorch, a popular ML package for Python, to train and output the models.

The Package requires a significant amount of setup before a user can begin training an agent. A user must first install the package into the unity project containing their game. The python package for Unity ML agents must then be installed as well as other prerequisite packages; such as *TensorBoard* and *PyTorch.* The user must then modify the entity whose behaviour will be trained to include components introduced by the ML Agents package. This includes overriding a class, Agent, the implementation given (ABOVE) is exemplary of the complexity of implementation and necessary understanding of the ML methods used in training when implementing these overrides. The classes overridden also require tuning outside of the scripts written. The Editor window provides an interface to assign parameters such as; maximum episode length in physics updates, the number of observations inputted and actions outputted, and the model used for determining the behaviour of the agent after training is completed and development can progress. The user must then create a .yaml file, specifying the hyperparameters used by the machine learning model throughout training. This would be an intimidating task for those unfamiliar with machine learning. While the user can access example .yaml files given in sample training environments and gain some familiarity through experimentation with this, the effect of the hyperparameters are explained briefly in the documentation for the package. After they have specified the hyperparameters, the user can then begin training their model by using the command prompt and specifying the location of the executable for their game environment or simply using the unity editor.

-Quote from SE paper, Buliding a toy example is trivial compared to production ready

Cleaning data. Cannot remove bad examples, hurts workflow from the main ML SE paper

On action received function does too much, should be smaller functions.

Learning env create new documentation. Multiple training areas within the same scene. Documentation out of date.

# Chapter 5

# Conclusions

The results from this project indicate that ...

**Chapter 6**

# Reflective Analysis

The project went well ... ha cool joke

# References

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B. and Zimmermann, T. (2019) Software engineering for machine learning. In: Anonymous *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 27/5/2019. Montreal, Canada: IEEE Press. 291-300.

Arpteg, A., Brinne, B., Crnkovic-Friis, L. and Bosch, J. (2018) Software Engineering Challenges of Deep Learning. In: Anonymous *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 29/8/2018-31/8/2018. IEEE. 50-59.

Beeching, E., Debangoye, J., Simonin, O. and Wolf, C. (2021) Godot Reinforcement Learning Agents. Available from <https://arxiv.org/abs/2112.03636> [accessed 16 February 2022 2022].

Chen, Z. and Yi, D. (2017) The Game Imitation: Deep Supervised Convolutional Networks for Quick Video Game AI. Available from <https://arxiv.org/abs/1702.05663>.

Dahlskog, S. and Togelius, J. (2012) Patterns and Procedural Content Generation : Revisiting Mario in World 1 Level 1. In: Anonymous *Proceedings of the First Workshop on Design Patterns in Games*, 29 May 2012. New York: ACM.

de Haan, P., Jayaraman, D. and Levine, S. (2019) Causal Confusion in Imitation Learning. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alch\'{e}-Buc, E. Fox and R. Garnett. *33rd Conference on Neural Information Processing Systems*, 8-14 Dec 2019. Vancouver, Canada: Curran Associates, Inc.

Dodge Roll. (2016) *Enter the Gungeon.* Austin, USA:.

Fullerton, T. (2018) *Game Design Workshop: A Playcentric Approach to Creating Innovative Games.* 4th. Boca Raton, Florida: CRC Press.

Johnson, H.A. (2017) Trello. *Journal of the Medical Library Association,* 105 (2) 209. Available from <https://explore.openaire.eu/search/publication?articleId=doajarticles::e47c38a08a2992f70308d50717da6c21>.

Karakovskiy, S. and Togelius, J. (2012) The Mario AI Benchmark and Competitions. *IEEE transactions on computational intelligence and AI in games.,* 4 (1) 55-67. Available from <https://ieeexplore.ieee.org/document/6156425>.

Krumins, A. (2019) *MindMaker: Deep Reinforcement Learning.* [accessed 15/5/ 2022].

Lee, G., Min Luo, Zambetta, F. and Xiaodong Li. (2014) Learning a Super Mario controller from examples of human play. In: Anonymous *2014 IEEE Congress on Evolutionary Computation (CEC)*, July 6-11. Beijing, China: IEEE. 1-8.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013) Playing Atari with Deep Reinforcement Learning. Available from <https://arxiv.org/abs/1312.5602>.

Risi, S. and Togelius, J. (2020) Increasing generality in machine learning through procedural content generation. *Nature machine intelligence,* 2 (8) 428-436. Available from <https://arxiv.org/pdf/1911.13071.pdf>.

Sommerville, I. (2011) *Software engineering.* 9. ed. Boston, Massachusetts: Pearson.

Togelius, J., Shaker, N., Karakovskiy, S. and Yannakakis, G.N. (2013) The Mario AI Championship 2009-2012. *The AI magazine,* 34 (3) 89-92. Available from <https://search.proquest.com/docview/1443701747>.

Torrado, R.R., Bontrager, P., Togelius, J., Liu, J. and Perez-Liebana, D. (2018) Deep Reinforcement Learning for General Video Game AI. In: Anonymous IEEE. 1-8.

Tufano, R., Scalabrino, S., Pascarella, L., Aghajani, E., Oliveto, R. and Bavota, G. (2022) Using Reinforcement Learning for Load Testing of Video Games. *ArXiv,* abs/2201.06865.

Unity Technologies. (2022) *Agents.* Available from <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#rewards-summary--best-practices> [accessed 18/05/ 2022].

Unity Technologies. (2021) *Example Learning Environments.* Available from <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Examples.md>.

van der Linden, R., Lopes, R. and Bidarra, R. (2014) Procedural Generation of Dungeons. *IEEE transactions on computational intelligence and AI in games.,* 6 (1) 78-89. Available from <https://ieeexplore.ieee.org/document/6661386>.

Zou, H., Ren, T., Yan, D., Su, H. and Zhu, J. (2019) Reward Shaping via Meta-Learning. Available from <https://arxiv.org/abs/1901.09330>.