**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**

[1. Introduction 1](#_Toc104194407)

[1.1 Some notes 1](#_Toc104194408)

[1.2 Testing some mathematics 1](#_Toc104194409)

[1.3 Undergraduate Project Report 2](#_Toc104194410)

[1.4 Referencing 2](#_Toc104194411)

[1.4.1 Ludography 2](#_Toc104194412)

[2. Literature Review 3](#_Toc104194413)

[2.1 Background 3](#_Toc104194414)

[3. Methodology 8](#_Toc104194415)

[3.1 Software Development 9](#_Toc104194416)

[3.2 Project Management 12](#_Toc104194417)

[3.3 Toolsets and Machine Environments 14](#_Toc104194418)

[3.4 Research Methods 16](#_Toc104194419)

[4. Design, Development and Evaluation 18](#_Toc104194420)

[4.1 Requirements 18](#_Toc104194421)

[4.2 Game Design 21](#_Toc104194422)

[4.2 Unity ML Agents Integration 22](#_Toc104194423)

[Agent.CollectObservations(VectorSensor sensor) 23](#_Toc104194424)

[Agent.OnActionRecieved(ActionBuffers actions) 23](#_Toc104194425)

[Observations 24](#_Toc104194426)

[Training Environment Instances 25](#_Toc104194427)

[Demonstrations 25](#_Toc104194428)

[Hyperparameters 25](#_Toc104194429)

[Experimentation and Tuning of ML Model 33](#_Toc104194430)

[Initial Tests 33](#_Toc104194431)

[Complex Level RL Testing 36](#_Toc104194432)

[4.4 Agent 40](#_Toc104194433)

[4.5 Analysis of Agent and Final ML Methods 40](#_Toc104194434)

[4.6 Critique of Unity ML Agents 40](#_Toc104194435)

[5. Chapter 5 42](#_Toc104194436)

[6. Conclusions 42](#_Toc104194437)

[7. Reflective Analysis 43](#_Toc104194438)

[8. References 44](#_Toc104194439)

**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 agents whose behaviour improves the player experience is an important part of creating engaging games (Fullerton, 2018, 111). Imitation Learning and Deep Reinforcement Learning can be used in combination to create AI that controls these agents. Imitation Learning (IL) attempts to recreate behaviour, enacting the same or equivalent action as it’s trainer when given the same stimulus (Torrado et al., 2018, 1). Reinforcement Learning (RL) agents interact with an environment repeatedly, using a reward function to gauge the success of a single action or combination of actions (Sutton and Barto, 2018).

## These techniques have been used in combination to produce AI which can play games. Performing IL as a pre-training step before RL has been seen to result in a significant reduction in training time in complex 3D gameplay, even surpassing the expert it trained from (Harmer, et al., 2018). The combined use of these techniques means that effective AI can be produced rapidly. This would make the creation of AI easier for developers, as this would allow AI to be trained by players/play testers. In future, these techniques could be used to imitate a player directly allowing players to compete with players who are offline making features like Drivatars of Forza Horizon and Forza Motorsport games simpler to create (Playground Games, 2021).

## In this project, a combination of IL and Deep RL techniques will be implemented to train an AI to navigate a 2D side-scrolling level. Side-Scrollers are a typical testbed for ML in game AI with Super Mario Bros. often used to test new techniques (Lee, et al., 2014). Reinforcement Learning will be completed on a procedurally generated level in order to reduce the risk of overfitting, which when pre-training though IL, is a significant risk (Harmer et al., 2018).

**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. 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 (Schulman et al., 2017). This is implemented 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, 1) 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 behaviour 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 (Camargo and Sáenz, 2021, 1). 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, 1). IRL derives a reward function from the behaviour 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, 1).

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 behavioural 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. 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 analysing 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 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

## 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, 2021a). 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.

## 3.2 Project Management

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

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

RISKS

## 3.4 Benchmarks and Evaluation 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).

**Chapter 4**

# Design, Development and Evaluation

## 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.

## 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

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.

## 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 2: 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 3: Average Reward During Training for the Experimentation of Progress Reward |
|  |
|  |

### 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 4: Mean Reward earned During Training of the Simple RL Model |
| Chart, line chart  Description automatically generated |
| Figure 5: 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. 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 Technologies, 2021c). The curiosity module rewards the model for visiting states which it has not seen before. This should be balanced with the reward gathered from the environment such that the agent is encouraged to complete a level above finding an entirely new state. In their proposal of a curiosity mechanism, Pathak et. al. use pixel input from *Super Mario* without an environment defined reward (2017, 488). Their agent progressed through the level until it found the first gap which required a specific set of inputs to traverse. The agent never successfully completed this jump so never discovered that there was more level passed this point, without the ability to generalise from previous experiences. It was found that the modification of the curiosity strength did not improve quality of agent produced, with the original value of 0.02 providing best behaving agent and most stable training.

|  |
| --- |
| Chart  Description automatically generated with medium confidence |
| Figure : Reward During training with Different Curiosity Hyperparameters |

After several more unsuccessful attempts with hyperparameter tuning, focus shifted to modifying the reward function and observations to improve the set of inputs available to the model. This was done in two ways, changing the punishment to the agent for falling off the level and though changing the observation of position and altering the reward function to obscure the position of the agent from itself.

A frequent occurrence in failed tests was the resulting agent tending to jump backwards off the level IMAGE as mentioned previously. The agent would spawn into the level and immediately jump off as quickly as possible. The agent would not progress further through the level after this. It was theorised that this failing was caused by the negative reward of -1 given for falling off the stage. The Unity ML documentation states “Excessive negative rewards can result in the agent failing to learn any meaningful behavior ” (Unity Technologies, 2021c). The agent earned a reward of -1 no matter where the agent fell in the level, meaning falling in the final chunk would garner the same reward as falling in the first. The size of the negative reward could cause the model to associate the actions of progression to the action of failure, thus causing the agent to move away from this behaviour. To remedy this, the reward function was altered to reduce the negative reward for falling during a level to -0.2.

|  |
| --- |
|  |
| Figure : New Reward Function for Falling |

The observations laid out in IMAGE OF OBVS includes the current position of the agent in relation to the start of the level. Upon further consideration, the position of the agent was deemed unnecessary to the learning of a generalised agent. This had not presented an issue in the previous successful learning attempt as the agent had learned to navigate a single, unchanging level. It is possible the agent had learned when to jump based on its position rather than the information gathered from ray casts and the other observations.

The removal of this observation could have led to the obscuring of the relationship between progress through the level and higher reward. To reintroduce this connection, the reward function was altered to improve the calculation of reward for the agents position in the level. The position of the agent was analysed after a timestep which was parameterised and accessible from the Unity Editor. If the position of the agent was greater than it had been at any analysis previously, the agent was rewarded with a percentage of a small reward proportional to the agent’s progress through the level. This allows the agent to learn that progress is desirable, without linking actions to the agents position in the level.

|  |
| --- |
| Graphical user interface, text, application, email  Description automatically generated |
| Figure : Calculation of Reward for Distance Travelled |

Several combinations of frequency and maximum size of reward were tested. Those with an overly frequent or too great reward failed to converge to create an agent which can complete the complex level. These agents were observed to move through the level slowly which implies the agent has learned to maximise the incorrect value of the function, suggesting the reward for progressing was stronger than the reward for completing the level. The combination of 0.02 max reward and 10 steps between granting of reward resulted in an agent which could successfully navigate the complex level consistently.

|  |
| --- |
| Chart, line chart, histogram  Description automatically generated |
| Figure : Reward During Training for Tuning of Distance Travelled |

### Imitation Learning Agent on a Complex Level

An IL agent was trained using the PPO parameters from one of the Unity ML training environments. None of the tuning of hyperparameters during the experimentation with RL had resulted in an agent which could complete the level when the reward function or observations had been the cause of the failure of the model to converge on a desirable solution, with only the curiosity function resulting in a noticeable change in behaviour. The model now used GAIL as well as PPO to explore the available actions and maximise the reward function, while also trying to maximise the reward earned through the imitation of the demonstrations. Demonstrations were recorded of a human completing the level. Around 5 minutes of demonstration data was used as a training set for the model. The performances recorded did not vary a huge amount, with the same actions being taken for the majority of demonstrations so as to not add conflicting actions to the training set. The model was successfully able to learn to complete the complex level with a change from the original hyperparameters in setting the GAIL hyperparameter *use\_actions* to true. This allowed the model to view the inputs from the demonstrations allowing a faster, more accurate knowledge of the consequences of actions (Unity Technologies, 2021c). The performance of this agent was compared to the performance of the successful RL model from earlier and the results are shown in Figure 9 and Figure 10.

|  |
| --- |
| Chart, line chart  Description automatically generated |
| Figure : Episode Length Comparison between RL and IL model on Complex level |
|  |
| Figure : Episode Length Comparison between RL and IL model on Complex level |

The IL model and RL model appear to converge on a solution close to that of their final solutions at a similar time, with both functions seemingly finding a close to optimal solution within the first 500000 training steps. The models produce agents which display very different behaviours, however. This can be seen in the disparity between the average lengths of episodes displayed at the end of training in Figure 10, where the IL model takes around 2.9k steps to complete a level where the RL model takes less than 1000. 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 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 as the agent would risk falling through a gap in the level if it used this without knowledge of the next section. This was demonstrated in the next, more complex level where this model could not regularly complete episodes due to falling through gaps.

The IL agent did not learn this exploit and completed the level at a slower pace, not exploiting the dash or double jump to move faster. This behaviour was displayed in the demonstrations, implying that the model optimises for proximity to demonstrated behaviour over maximising the reward earned. This is seemingly conflicting with the higher reward earned by the IL function throughout training, though this is likely due to the reward function component shown in Figure 7 which has a greater impact on the total reward than previously considered. The RL agent moving faster yet gaining less reward and the near complete stabilisation in reward toward the end of its training suggests that the RL agent attempts to collect a big reward as quickly as it can, ignoring the overall negative effect of ending the episode early over delaying. This is also supported by the behaviour of jumping off the level discussed earlier.

### Imitation Learning on a More Complex Environment

The IL model was trained on a more complex environment, this time including a double jump and numerous gaps to traverse. The agent could not converge on an effective solution to completing the level. The agent was observed to move at an incredibly slow pace through the level, only managing to navigate around half the distance if the agent were successful in jumping the gaps, which it would often fall through due to its slow movement and reluctancy to use the dash function. It was thought the strength of the GAIL signal may be causing the undesirable behaviour. If the strength of the GAIL signal were too low, the agent may not mimic the demonstration well enough leading to the exploitation of the reward signal. If the GAIL signal were too high, the agent would maximise this signal above exploring to find a more optimal solution (Ho and Ermon, 2016). Several values for the GAIL strength hyperparameter were tested, none of which improved on the behaviour shown in the original test, Figure 13. Also displayed in Figure 13 is a purely RL model (0.0 GAIL strength) attempt to learn the level. This model, as mentioned previously, found the wall jump exploit and moved at rapid pace. This resulted in an occasionally successful agent, which displayed a high risk/high reward behaviours, often falling through the gaps in the level. When observed completing a level with a greater maximum time steps per attempt than their training, the IL models were able to, somewhat consistently, complete the level. This supports the theory that the agents were attempting to maximise their reward by moving slowly through the level.

|  |
| --- |
| Chart  Description automatically generated |
| Figure : Second Complex Level |
|  |
| Figure : Mean Reward during Testing of GAIL Strength |

To prevent the agent from moving slowly through the level the reward function was again modified. The reward given to the agent for progressing through the level was altered to scale inversely with the time taken by the agent to reach that point. This led to a small increase in the speed of the agent however it still could not reliably complete levels. In order to encourage the agent to act as the demonstrations showed, another form of imitation learning was added to the model. Behavioural cloning (BC) has been used to create an agent which can play the first-person shooter *CSGO*. Pearce and Zhu found that BC allowed the creation of an agent which could beat the medium AI found in the game and exhibited some human like behaviours, such as pausing slightly to “readjust the mouse” (2021). BC is used as a pretraining step in the Unity ML package, allowing the agent to learn directly from the inputs of the demonstration. This can be done over a short period of steps (in this case 100000) and then GAIL and RL are used to optimise the agent. This resulted in a successful agent, which did not exhibit the slow pace of the original and completed the level consistently.

|  |
| --- |
| Graphical user interface, text  Description automatically generated |
| Figure : Reward Function Modification for Scaling Reward |

### Generalising the Side Scroller Agent

In attempting to create an agent which could perform well on any level presented to it, several combinations of RL and IL methods were used. Each of the IL methods were trained using the same demonstration data, which consisted of around 15 minutes of recordings on a different set of test levels to those presented to the models during training. The reward earned by these agents is presented is presented in Figure 15. The various IL agents were observed to complete parts of a level well, however often failed when faced with a particular type of double jump or a large gap. None of the methods produced a reliable agent for completing a level, shown in the lack of stability in reward function. The pure RL model was the worst performing agent, suggesting that RL alone is not a suitable method for training an agent to perform tasks on a generalisable environment.

|  |
| --- |
| Graphical user interface, chart, line chart  Description automatically generated |
| Figure 15: Reward Earned by IL/RL Models During Training |

As previously, the reward function was altered to try to improve the results of the agents. Up until this point, the reward function did not give a negative reward to the agent for taking too long in a level, instead relying on a reduction in the magnitude of the reward function to incentivise positive behaviour. A negative reward of -1 was given to the agent if it timed out of the level, therefore encouraging exploration and falling through a gap above simply waiting for the episode to end. This change had the potential to significantly reduce the quality of an agent produced if the punishment were too high, however the reward gained through exploration in the level was considered likely to prevent this problem and incentivise positive movement.

|  |
| --- |
| Text  Description automatically generated |
| Figure 16: Reward Function Modification to Give Punishment for Non-Completion |

This reward function was tested on a RL + BC model and a model using RL + BC + GAIL. These models were also unsuccessful, though the latter showed improvement over those tested previously behaving in a more similar manner to the demonstrations and completing double and dash jump obstacles more frequently. In the majority of levels the model was unsuccessful. This should have been provable via the level completion metric mentioned earlier, however this metric failed recording sporadically during training and not recording any data for the final 2 million steps. A similar issue was found with the jumps completed metric. Due to issues with the real world time of training, discussed later, this was the last model produced.

|  |
| --- |
| Chart, line chart, histogram  Description automatically generated |
| Figure 17: Performance of Final Models |

### Evaluation of Testing

Training agents was successful for single levels no matter their complexity. Significant modification of the reward function was required to achieve an agent which could complete each level, however. Several combinations of IL methods were tested in the attempts to create an agent which had a general solution, however none of these could be considered successful. Further tuning of hyperparameters such as GAIL and BC strength could help improve the performance of this agent. A lower value should result in an agent which attempts to mimic the exact actions of the demonstrations less, a desirable quality in generalisation. The difficulty in the tuning of agents at every stage highlights the difficulties faced in ML of the complexity of agent required to execute even relatively simple tasks and the explainability of systems.

## 4.6 Critique of Unity ML Agents

The functionality of Unity ML Agents is divided into 3 interfaces: the unity editor, C# classes which can be overwritten, and a command line interface. 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 into the user’s environment 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 the Agent class, which can be a laborious and complex process an example of which has been discussed at length. The classes of the package and those 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 when the agent is not being trained.

|  |
| --- |
| Graphical user interface  Description automatically generated |
| Figure 18: Some Parameters for Training Found in Unity ML Agents Components |

The user must then create a .yaml file, specifying the hyperparameters used by the machine learning model throughout training. While the user can access example .yaml files given in sample training environments and gain some familiarity through experimentation, this would be an intimidating task for those unfamiliar with machine learning. 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. The number of steps required to setup training could be prohibitive for users looking to use an ML method to train the behaviour of their agents. The depth of knowledge needed of ML models for effective use in toy examples is also beyond what may be considered “simple” (Unity, 2022) as described in the package’s description which would make a production ready implementation near impossible for users new to machine learning due to the added complexity described by Arpteg et. al. (2018, 50).

The package does not provide a method through which to remove bad demonstrations from recordings made by a developer. This makes providing good quality demonstrations for IL agents more difficult and hinders the machine learning workflow described by Amershi et. al. and employed in this project (2019, 291).

The OnActionReceived() function described earlier could be considered to break the SOLID principals often used in software development. This function could be decomposed into several, reducing the potential for confusion when a user overrides the Agent component.

The documentation of the package is sparse once one progresses passed the introductory activities. The documentation is also often incorrect, describing an “Academy” component which cannot be accessed through the Unity Editor *Add component* menu, but is now a singleton only instanced and accessible during runtime. The documentation describes a set of parameters which can be used in the execution of training, one of these is *time\_scale (Unity Technologies, 2021b)*. This parameter should increase the rate at which “real time” is executed by the model, allowing more steps to be computed per second without interference with physics. This feature would not work on any device attempted and is a documented issue in the and as such all training took place at real time. This greatly hindered the progress of the project as a single training session could take upwards of 5 hours to complete.

While the Unity ML Agents package provides a useable implementation of ML techniques for the Unity Engine, the complexity of the systems exposed to a user as well as issue with both bugs and documentation make the package considerably inaccessible for developers new to machine learning.

# Chapter 5

# Conclusions

This project shows that an IL agent can be used to complete a side-scrolling level, though further development of the agent created would be needed to produce an agent which can generalise to complete any level it was faced with. The development of a simple side scroller inspired by *Infinite Mario* is documented and the procedural generation of its levels is described. The side scrolling game created for this project was successful, providing a consistent training environment for the models to learn. The implementation of the Unity ML Agents package is described, with key functions highlighted and the processes required to begin development an agent discussed.

The development of the final agent presented took considerable tuning of the reward function and experimentation with combinations of ML methods. The best of these appeared to be RL + BC + GAIL, so future work should first develop on this agent. Throughout Section 4 the behaviour and consequences of modifications to the reward function are described. Several features were added to the reward function as development progressed. while the observations were simplified, removing the knowledge of the agents position from the model. The failure of some of the considered improvements to improve the model exemplifies the difficulties in explainability in machine learning, though the observational observations possible through ML in games makes the visualisation of issues more simple than typical ML problem. The Unity ML package hindered the progress of this project and some of the issues are addressed. Had the time scale altering been functional, more tests would have been completed and possibly a successful agent would have been made.

**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].

Camargo, J.E. and Sáenz, R. (2021) Evaluating the impact of curriculum learning on the training process for an intelligent agent in a video game. *Inteligencia Artificial,* 24 (68) 1-20. Available from <https://doaj.org/article/4e7ab9b41f454cc8a2676666fe84a211>.

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.

Ho, J. and Ermon, S. (2016) Generative Adversarial Imitation Learning. In: D. Lee, M. Sugiyama, U. Luxburg, I. Guyon and R. Garnett. *Advances in Neural Information Processing Systems*, 5-10 Dec 2016. New York, USA: Curran Associates, Inc.

J. Harmer, L. Gisslén, J. del Val, H. Holst, J. Bergdahl, T. Olsson, K. Sjöö and M. Nordin. (2018) Imitation Learning with Concurrent Actions in 3D Games. In: Anonymous *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 14-17 Aug 2018. New Jersey: IEEE. 1-8.

Jacob, M., Devlin, S. and Hofmann, K. (2020) "It's Unwieldy and It Takes a Lot of Time." Challenges and Opportunities for Creating Agents in Commercial Games. In: Anonymous *Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, October 19–23, 2020. Palo Alto: California: AAAI 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.

Liu, H., Chen, Y., Tan, J. and AngJr, M.H. (2021) Improving Learning from Demonstrations by Learning from Experience. Available from <https://arxiv.org/abs/2111.08156>.

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>.

Pathak, D., Agrawal, P., Efros, A.A. and Darrell, T. (2017) Curiosity-Driven Exploration by Self-Supervised Prediction. In: Anonymous *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, July 21 - July 26 2017. Honolulu, USA: IEEE. 488-489.

Pearce, T. and Zhu, J. (2021) Counter-Strike Deathmatch with Large-Scale Behavioural Cloning. Available from <https://arxiv.org/abs/2104.04258>.

Playground Games. (2021) *Forza Horizon 5.* Redmond, USA:.

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>.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017) Proximal Policy Optimization Algorithms. *ArXiv,* abs/1707.06347.

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

Sutton, R.S. and Barto, A.G. (2018) *Reinforcement Learning.* Second. Cambridge, USA: MIT Press.

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. (2022) *Unity Machine Learning Agents.* Available from <https://unity.com/products/machine-learning-agents> [accessed 16 February 2022].

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

Unity Technologies. (2021b) *Python API.* Available from <https://github.com/Unity-Technologies/ml-agents/blob/f9ec4bad16c3ce9765a160bb0276994ddd8eaddf/docs/Python-API.md>.

Unity Technologies. (2021c) *Training Configuration File  
.* Available from <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md> [accessed 20/5/ 2022].

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>.