Literature review: Side Scrollers

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).

In the design of the Side Scrolling game produced through which to train the agents, a level of complexity similar to that of Lee et al. produced in their simplified version of *Super Mario* mentioned earlier (2014). The player can navigate the level through moving left and right as well as jumping, double jumping, a mid-air dash, and a stomp. These actions provide the player with several options as to how they navigate levels, allowing some obstacles to be solved with more than one solution e.g. a player could double jump over a small gap or dash jump over it. This can be used to assess how closely an agent mimics a player’s style as the behaviours can be recorded and compared.

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 (Revisiting Mario in level 1-1). 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.

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, ????). 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, ????) 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 taken into account. 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 a 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 along the same level using demonstrations of a 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 an approximation 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 produce it was noted that… 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.

|  |
| --- |
| Graphical user interface  Description automatically generated |
| Figure 1: Mean Reward of Test 36 and 41 during training |
| Graphical user interface, chart  Description automatically generated |
| Figure 2:Average Length of Episode Across Training during Test 36 and 41 |

Curriculum learning was used to train the

Best practices for unity rewards encourage adding a small reward for locomotion tasks.

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* must also then be installed. 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 obeservations 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.

Complex setup, multiple python envs need to be setup

Recommendations for which hyperparameters to change in certain situations.

Time scale does not work

Must be activated from a cmd window

Separate to the unity editor

.yaml file cannot be modified within the editor, meaning there is a divide in where the model is parameterised (stacking done in editor)