Universiteit Leiden
The Netherlands

Data Science and Artificial Intelligence

Exploring algorithmic approaches to best the game Hill Climb Racing

Alex Zheng

Supervisors:
Matthias Müller-Brockhausen & Evert van Nieuwenburg

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

03/07/2024

Abstract

This thesis explores a reinforcement learning (RL) environment inspired by the game *Hill Climb Racing* (HCR), incorporating diverse observation spaces, action spaces, and reward functions. While no research currently exists on RL algorithms applied to HCR, similar studies have been conducted on the Mountain Car environment. The aim is to develop an RL agent that maximises its score, measured by the distance travelled, in an HCR-like environment. Moreover, the thesis evaluates multiple action spaces, including a discrete action space with three actions and a continuous action space, alongside various reward functions such as distance-based and wheel speed-based rewards. The Proximal Policy Optimization (PPO) algorithm was utilised to train the most promising agents. Results of the evaluation experiments demonstrate that an agent utilising an aggressive wheel speed-based reward function within a continuous action space performed best, achieving a mean score of 773 in the standard environment. This agent further excelled in environments with increasing difficulty, consistently reaching the maximum score of 1000 after 200,000 training timesteps.

Contents

1 Introduction

1.1 Motivation

Reinforcement learning (RL) is a category of modern and state-of-the-art machine learning algorithms and is one of the most active research topics in the artificial intelligence field, especially in the last three decades [1]. When a reinforcement learning agent is in an environment, it can only observe a certain quantity of observations. However, our human brains are capable of processing multiple sources of information and integrating them into a coherent representation of the world [2], and it is able to determine the correct actions based on the observations. For example, a human driving a car who traverses through a road intersection is capable of observing the red, green, or orange traffic lights and a variety of vehicles and pedestrians. The human crosses the road and chooses the right time to brake or accelerate by utilising all those aforementioned observations. Nevertheless, choosing the correct actions to cross the road is challenging for a computer, such as a reinforcement learning agent, especially in environments with continuous action and observation spaces. The curse of dimensionality explains the difficulty these reinforcement learning algorithms often have. When the number of observations and actions (state-action pairs) in an environment rises, it becomes increasingly more convoluted for the agent to form an optimal policy because this state-action pair growth is exponential [3].

Additionally, one of reinforcement learning's core strengths is its ability to learn from its interaction with the environment. It learns optimal policies through interaction with the environment rather than from a fixed dataset. For example, learning from interactions helps discover the optimal path to achieve high scores in games. Moreover, RL is capable of balancing exploration (by trying new actions) and exploitation (using known actions to maximise rewards). Exploration and exploitation are beneficial because many games have a wide array of potential actions, and diverse actions are optimal in distinct circumstances.

Furthermore, games can be simulated repeatedly, allowing RL algorithms to train extensively, and simulations can be scaled up to run many parallel instances, accelerating the learning process. Also, there are many examples of games where RL has been applied and has achieved or even surpassed human-level performance, demonstrating its powerful learning capabilities. For instance, DeepMind's AlphaZero achieved superhuman performance in chess [4], and RL agents have mastered a wide array of Atari games. More is explored in Section 2 regarding these examples.

Hence, this thesis will explore a reinforcement learning environment similar to the popular game *Hill Climb Racing* (HCR). In this environment, numerous diverse potential observations, action spaces and reward functions influence the agent's learning in the environment. This environment can be used as a benchmark for reinforcement learning algorithms, yet this is not the primary objective of the thesis. Currently, there is no research on reinforcement learning algorithms applied to the game HCR. However, similar research has been done on a comparable game called *Mountain Car*, a standard classic control environment in the Gymnasium library [5]. Moreover, the environment will not be identical to the original HCR game but has core similarities; this will be explicated further in Section 3. Therefore, this thesis will examine the process of creating a reinforcement learning agent that tries to maximise the highest attainable score in our version of the HCR environment.

1.2 Research questions

As mentioned in the last section, the main research goal is to create a reinforcement learning agent that attempts to maximise the highest attainable score in the HCR environment. Multiple research questions will help us to achieve this goal. The main research question guiding this thesis is the following:

What algorithmic approach that is experimented with achieves the best-performing RL agent in the Hill Climb Racing environment?

A fair amount of research is needed to answer this question, and for this reason, many experiments are conducted within the environment. These experiments are supported by two secondary research questions that help us answer the main research question. The first secondary research question is formulated as described below.

What is the optimal reward design for an agent in the Hill Climb Racing environment?

The reward design is a key variable in the environment, as the reward design dictates how much the reward will be per step in the environment. The reward function heavily influences the agent's decision and performance [6]. Hence, experimentation is necessary to find an optimal reward function. Lastly, the additional secondary research question is as stated below.

How do different action spaces affect the agent's performance in the Hill Climb Racing environment?

The actions used by the agent in the environment should be defined and researched. To achieve our main research goal, different action spaces, such as discrete and continuous input, are experimented with to discover the best-performing agent.

1.3 Thesis overview

This section will provide an overview of the thesis. The current section, Section 1, contains the introduction of the thesis. Section 2 explores already established research literature related to our thesis. Furthermore, a background overview, which discusses the game HCR and the similar Gymnasium environment of Mountain Car, can be found in Section 3. Moreover, Section 4 explicates how the environment is built up and what algorithms are used in the experiments. The experiments can be found in Section 5, which also includes the results. Eventually, the thesis ends with the conclusion and further research in Section 6.

2 Related Work

Firstly, Q-learning is a reinforcement learning technique based on a state-action table with a Q-value, whereas DQN uses neural networks with ER to calculate the Q-value for an action. In 2020, Vu et al. applied Q-learning and SARSA with a ϵ-greedy policy and a CNN for observations to the game Flappy Bird [7]. SARSA is similar to Q-learning but uses on-policy learning, whereas Q-learning is off-policy. As Q-learning encounters the curse of dimensionality in a continuous observation space such as the game Flappy Bird, the researchers realised that discretisation (reducing the number of states to explore) of the environment was necessary. Without the discretisation, Q-learning could not achieve a mean score of 1 since there are 288 x 512 x 16 states to explore. In contrast, by discretising the number of states to 10 x 10 x 16, the researchers achieved a mean score of 209 using Q-learning and backward Q-value updates. In 2013, Mnih et al. [8] applied deep Q-learning with experience replay combined with a convolutional neural network for observations on seven different Atari games; the algorithm the researchers utilised is known as Deep Q-Networks (DQN). They outperformed six out of the seven games compared to previous methods, and in three games, the DQN outperformed human experts. Of the seven games, Enduro is the most similar to HCR. This is a racing game where players must pass different cars on their racing path while continuously racing as fast as possible. Enduro was one of the games in which the performance was better than that of an expert human player.

In 2017, a study was done by Google DeepMind on chess, shogi and Go, where David et al. created the AlphaZero algorithm [4]. The AlphaZero algorithm had no access to any domain knowledge except the game's rules. It implemented RL at such a level that it achieved superhuman performance in these games. AlphaZero uses a Monte-Carlo tree search (MCTS) to simulate games with a deep neural network as its evaluation function. At the end of each simulated game, a self-play RL algorithm updates the parameters of the deep neural network evaluation function. It was able to outperform any state-of-the-art chess engine at the time. However, it is clear that chess, Go, and shogi are games with a discrete environment, meaning a finite number of actions can be performed within the game. HCR is a game with a continuous observation space (observation space with floating point(s)) and, in our environment, also a potential continuous action space. Thus, additional research on games with continuous observation spaces or continuous action spaces should be explored.

In 2019, OpenAI [9] published an article regarding the utilisation of a deep reinforcement learning AI system to compete in Dota 2. Dota 2 is a popular multiplayer online battle arena (MOBA) esports game with a continuous environment, specifically, a high-dimensional action space and observation space. OpenAI Five defeated the world champion team of the game and was the first AI system to do so in an esports game like Dota 2. The AI system has demonstrated this accomplishment by using self-play reinforcement learning and showed how AI systems can perform at superhuman levels under challenging tasks. The policy used in the AI system was trained using PPO, similar to the study conducted by Holubar et al. [10] and our own agent. This was combined with a long short-term memory recurrent neural network (LSTM) that outputs a value function. The policy is trained using collected self-play experience, similar to how AlphaZero uses self-play to train. Moreover, a study by Holubar et al. explored a novel racing environment with continuous action and state spaces [10]. Agents in this racing environment were required to control the steering and acceleration of a car while driving on a randomly generated race track. Analogous to the HCR agent in our experiments, the agent uses the proximal policy optimisation (PPO) algorithm to train

its policy. The researchers also used the sampled policy gradient (SPG) algorithm and compared the agents' performance with SPG to PPO. The researchers also studied the effects of experience replay (ER) in both algorithms. PPO showed proficiency in capturing new information in continuous action spaces. Nevertheless, they discovered that ER is not as beneficial for PPOs in continuous action spaces as it is for SPG. All versions of SPG outperformed PPO when ER was used, showing that SPG can be favourable for continuous-action reinforcement learning. Lastly, in 2023, the thesis of Lex Janssens used PPO on a self-made version of the game Crossy Road [11]. The action space in his environment is discrete and has many observing states similar to HCR. The thesis concluded that the size of state representation is essential for a given environment. Agents in the game with a small state representation did not perform as well as agents with a larger state representation. The researcher also discovered that heuristic rewards for actions increased the agent's performance.

In conclusion, RL algorithms experimented on games, such as HCR, are not an unexplored domain. The research shows that PPO is the best-performing algorithm in continuous action spaces and continuous observation spaces. Thus, PPO is applied to the HCR environment in our experiments because HCR's observation space is continuous, and the action space is both discrete and continuous. Furthermore, research also showed a discrete action space with grid discretization for the observations shows promising results, yet, this will not be applied in our experiments.

3 Background

3.1 Mountain Car problem

A classic control problem that has existed in reinforcement learning for a protracted time is the mountain car problem. The problem environment consisted of a car and a mountain and was first mentioned in Andrew Moore's PHD thesis in 1990 [12]. Later on, Sutton mentioned this control problem in his paper about generalisations in reinforcement learning. In this paper, he demonstrates how the learning curve converges using the SARSA algorithm and sparse-coarse-coded function approximators for the mountain car problem and other classic control problems [13]. However, the mountain car problem became more popular after Sutton and Barto added it to the well-known *Reinforcement Learning: An Introduction* book in 1998 [14].

In the problem environment, the car is placed stochastically at the bottom of the mountain valley and surrounded by steep hills on the front and back, see Figure 1. Its goal is to drive up to the top of the hill and reach the goal state, which can only be accomplished by using gravity and the surrounding steep hills. The Gymnasium library contains two recreated environments of the mountain car problem. One with a continuous action space representing an array ranging from -1 to 1 containing a float32 number that represents the force applied to the car. Furthermore, the second environment has a discrete action space containing three discrete deterministic actions. These three actions are acceleration to the left, not accelerating at all, and acceleration to the right. Two observation variables define the observation space in this environment: the car's position along the x-axis, which ranges from -1.2 to 0.6, and the car's velocity, which ranges between -0.07 and 0.07 [5].

Over the years, many different algorithms have been applied to solve mountain car problems. One of these is Q-learning. In 2022, researchers modelled the mountain car problem and solved it using Q-learning and SARSA with state discretization [15]. The position and velocity observations are discretized such that there is a finite amount (ranging from 10 to 40 in this paper) of the aforementioned variables, making the size of the Q table smaller and thus making it easier and faster for the learning curve to converge as the total number of states to explore is less. Note that the action space in this research was discrete. SARSA and Q-learning solved the problem fully (meaning 100% accuracy in test runs) depending on the amount of discretization. Similar results have been achieved by another study that also showed how tabular Q-learning with discretization was faster to train compared to SARSA and deep Q-learning [16]. Another intriguing RL method is to use human feedback to create the agent's behaviour. In 2011, Knox et al. demonstrated a paper that applied RL to the mountain car problem with the `TAMER` framework [17]. This framework incorporates human feedback to shape the agents' behaviour; the human feedback signals can be approval or disapproval, which a human teacher gives. The researchers combined the `TAMER` framework with SARSA to integrate human reinforcement signals and Markov decision process rewards in one fully integrated system and found it significantly improved the performance compared to the methods used individually.

Moreover, researchers in 2022 created an enhanced version of PPO. They introduced policy feedback (PPO-PF) [18]. This novel version of PPO has the Critic network estimate the value function and is used directly to perform a policy update. In contrast, the policy from the actor-network conducts the value function update, resulting in a collaborative update of both value and policy functions. PPO-PF was also applied to the mountain car from OpenAI Gym. They found that PPO-PF had a faster convergence speed, a more minor variance of rewards, and higher average rewards than the original version of PPO. The mountain car experiment showed an 18% improvement in the average rewards. However, the benchmarks on some Atari games demonstrate a decreased performance, such as on breakout with a -54.60% decrease in score compared to the original PPO and thus, the original PPO algorithm is still utilised in the experiments.

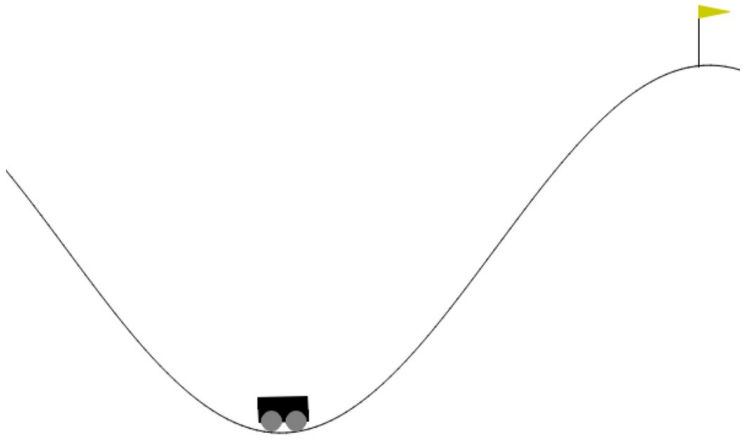


Figure 1: The Mountain Car Gymnasium environment [5].

3.2 Hill Climb Racing

Hill Climb Racing (HCR) is a popular 2D racing game developed by Fingersoft, first released in 2012 on IOS and Android and later in 2013 also on Microsoft Windows [19]. It won Finnish app of the year in 2014 and is currently the number 12 ranking racing game in the Dutch Apple App Store [20]. The game's goal is to drive a car as far as possible in a randomly generated world that contains hills that vary in height (the terrain). Similar to the mountain car problem, the car locates itself in a valley between hills. However, in HCR, the terrain is endless, and the height of the hills varies when driving further. The game's death condition is when either the player inside the car touches the car with their head or when the car runs out of fuel. Fuel is spawned throughout the level systematically. Coins and diamonds are likewise spawned through the level and can be used to upgrade the car. The player has access to the following two actions: the first action is increasing the gas to drive the car forward; during this action, the car rotates counterclockwise. The second action is reverse driving the car, which brakes the car in a forward motion. During a reverse drive, the car rotates clockwise.

As mentioned before, our environment version of HCR differs from the original HCR. The environment version uses Python, Box2D, Pygame and the Gymnasium API library. The game environment is based on a JavaScript version made earlier by a YouTuber named CodeBullet (also known as Evan G), which he published on GitHub [21]. In a video, he showcases an AI playing HCR. To explicate, he demonstrates how the NeuroEvolution of Augmenting Topologies (NEAT) algorithm can be used to play HCR. NEAT is an evolutionary algorithm that generates and evolves neural networks. It starts with simple networks and incrementally grows them into more complex structures. NEAT treats nodes inside the neural networks as genes and mutates them using crossover based on their performance in each new population's generation. CodeBullet's NEAT agent reached the end of his generated terrain at generation three using seven different observations that output to either gas or breaking [22]. Our custom HCR environment does not contain any fuel parameters and focuses more on the control problem of driving the car through rough terrain. Furthermore, the HCR environment uses a distinguishable terrain generation algorithm that is more rough compared to the original game. More about the ground generation is explored in 4.3. Additionally, the terrain does end at a certain distance in the experiments, unlike the endless levels in the original game. Coins and diamonds spawns are excluded, and the car has access to full power. Moreover, our environment includes different types of action spaces: discrete and continuous; more about the environment will be illustrated in Section 4.2. The HCR environment is published on GitHub and includes a human-playable mode.

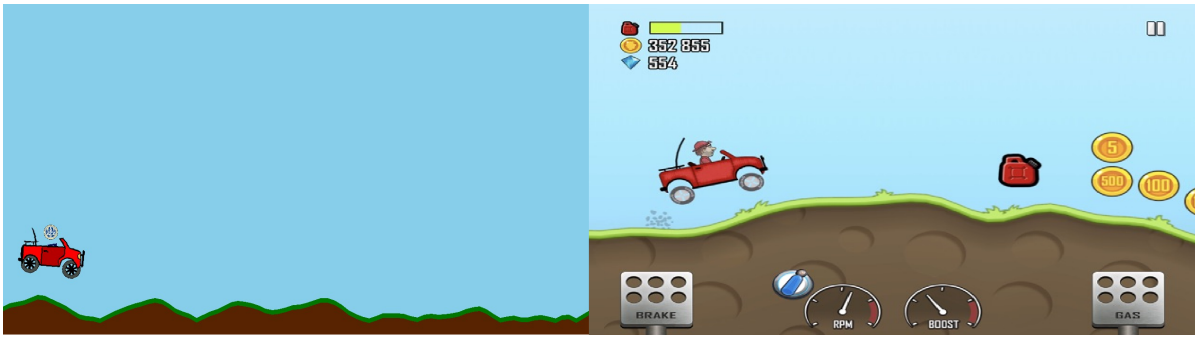


Figure 2: The HCR Pygame environment on the left and the original HCR game on the right [20].

4 Methods

4.1 Reinforcement learning

Reinforcement learning attempts to shape an agent's behaviour such that it is able to maximise cumulative rewards in its environment. Each observation combination can be seen as a state, and each state-action combination forms the state-action space. By observing the environment, the agent takes an action that positions the agent in a unique state where it receives a reward. The agent's behaviour or strategy is the decision process of taking an action based on the observations and rewards. In RL, the decision-making framework is called the policy, which tries to maximise an agent's total reward in the environment. Diverse policy learning methods exist to steer the agent in the right direction. In an on-policy method, the agent uses its target policy to perform actions in the environment and tries to improve its target policy based on the rewards and observations of those actions made by the target policy; an example of such an algorithm is SARSA. However, an off-policy method improves its target policy by using a different behaviour policy or another data source than its target policy, such as Q-learning [23]. Furthermore, one can classify RL algorithms into two different categories: model-based and model-free. Model-based RL relies on a model from the environment and uses this model to predict the outcomes of its actions as it can think in advance. Alternatively, model-free RL does not rely on a model but uses trial and error based on their experience [24]. Our reinforcement learning experiments will primarily use the Proximal Policy Optimization (PPO) algorithm, which is an on-policy and model-free reinforcement learning algorithm. More regarding PPO is explicated in Section 4.6.2.

4.2 Environment overview

As indicated previously, the environment is developed in Python and uses multiple libraries to assist. Box2D is utilised as the physics engine inside the environment, and Pygame renders the graphics. Furthermore, the environment operates the Gymnasium library for reinforcement learning functions such as resetting the environment, action, and observation spaces. More about the environment will be explored in the subsequent sections.

In the HCR environment, the agent embodies a person bound to a car. The game concludes with the character's contact with the ground while inside the car. This vehicle is provided with

dual motorised wheels (front and back) capable of operating in reverse or forward modes. The manipulation of motor speed can be executed through discrete actions, such as forward or gas commands, or through continuous actions, allowing the agent to set the motor speed according to its own policy.

Additionally, when the car is driven forward, it rotates counterclockwise, whereas in reverse, it rotates clockwise. This mechanic allows the agent to stabilise the car while airborne and navigating the terrain; however, the same mechanic likewise allows for mistakes that will conclude the game. The car, person, and wheels classes are all defined by Box2D shapes and rendered using Pygame. The complete code of these classes and the environment can be found on GitHub [25]. The environment resets itself every timestep when a death condition is reached and respawns the agent to its starting position, meaning a new episode starts. The starting position is a constant and hence does not change. However, the terrain generation is randomised in each new episode (with the same difficulty parameter). Lastly, when the agent is stuck (not travelling more than 20 metres) for more than 1200 timesteps (or frames), the episode will be truncated.

4.3 Terrain generation

The ground generation algorithm within the `Ground` class is implemented in the `randomize_ground` function. This function creates a procedurally generated terrain using Perlin noise, ensuring variability and a dynamic difficulty level as the player progresses. This section will explain how the core components generate the terrain, though it will not encapsulate every aspect. The complete code for the terrain generation algorithm can be found in appendix A.

The terrain incorporates coordinates (x, y) where x is the position along the x-axis and y is the position along the y-axis. These points are connected as they make up the vertices of a polygon; in our generation algorithm, they make up each section of the terrain. Each x-value is chosen using a smoothness factor that structurally spaces out the x-value until the maximum distance max_dist of the level is reached. For each x-value, an y-value is generated based on the steepness level S; the steepness level function uses linear interpolation and maps the x values to a range between the lower limit l and the upper limit u. In our environment, $l = 130$ and $u = 250$ are chosen to balance the difficulty by limiting the height extremes of the terrain. The steepness level function $S(x)$ is denoted by the following formula:

$$S(x) = l + \left(\frac{u - l}{max\_distance}\right) \cdot x$$

Where $max\_distance$ is the maximum distance of the level, $max\_distance$ is calculated by:

$$max\_distance = \texttt{MAX\_SCORE} + \texttt{X\_SPAWNING}$$

Where `MAX_SCORE` is the maximum score achievable in the level, and `X_SPAWNING` is the spawning location of the agent. Consider how S increases as the x-value increases, indicating a steepness increase in the terrain when the agent or player traverses through the terrain into the positive side of the x-axis.

The terrain is generated using Perlin noise [26] to ensure a degree of variability and dynamicity in the terrain. A Perlin noise function $P(z)$ from the Python noise library [27] is used to output N, which is the noise variable that will be utilised to choose a y-value for each x-value. For each S, the noise function $N(S)$ denoted by the following formula:

$$N(S) = \left| P\left(ground\_seed + \frac{x - flat\_length}{scale - S} \right) \right|$$

Where $ground\_seed$ is a seed randomly chosen using a uniform distribution between 0 and 100000 and $flat\_length$ is the initial flat part of the terrain where the agent or player spawns. $scale$ is a constant scaling factor that helps modulate the impact of S on the Perlin noise calculation, adjusting the frequency and, thus, the resulting pattern of the terrain. In essence, a higher scaling factor produces smoother noise with fewer abrupt changes, while a lower scaling factor produces noisier output, resulting in rougher terrain. For the scaling factor, $scale = 700$ was chosen as this results in a relatively balanced terrain.

Note how $N(S)$ calculates the absolute value from $P(z)$. The Perlin noise function in the Python library outputs numbers from $[-1, 1]$; when taking the absolute value from $P(z)$, it results in a terrain with a more irregular dynamic than the original game's terrain. However, the original Perlin noise function outputs values that range from $[0, 1]$. Figure 3 shows the difference between the two Perlin noise values.

Thereafter, S is used to calculate the maximum height parameter H_{max}, which defines the maximum height y can obtain. Comparable to the $S(x)$ function, a value is mapped using simple linear interpolation. The $H_{max}(S)$ function is defined as follows:

$$H_{max}(S) = \texttt{DIFFICULTY} + \left(\frac{b}{a}\right) \cdot S$$

Where `DIFFICULTY` is a predefined parameter at initialisation that modifies the terrain's difficulty at the method's initialisation. Furthermore, a and b are the range limit parameters for interpolation that are used to map values from a to b. The chosen parameters for our environment are `DIFFICULTY`=-150, $a = 200$ and $b = 320$. Note that the difficulty only increases until the steepness level reaches $a = 200$, while the upper steepness level limit is $u = 250$. This means after 80% of the terrain is traversed, the difficulty will not increase anymore as the maximum range limit of $a = 200$ is reached. However, in Section 5.5, an experiment is conducted with $a = u$, thus increasing the difficulty until the end of the terrain.

Finally, with the aforementioned variables x, N and H_{max}, it is achievable to calculate a ground vector G that contains the tuple coordinates (x, y). The function $G(x)$ assigns a height value y to all chosen x until the $max\_distance$ is reached to obtain the tuple (x, y). This function is defined as stated below:

$$G(x) = (x, \texttt{SCREEN\_HEIGHT} - (H_{max} - H_{min}) \cdot N)$$

Where H_{min} is a constant that guarantees a minimum ground height, and `SCREEN_HEIGHT` is a constant that represents the height of the screen. Note that $G(x)$ calculates the coordinate (x, y), where every coordinate calculated is added to an array that eventually represents the ground. The chosen values for these parameters are $min\_height = 30$ and the constant `SCREEN_HEIGHT`=720.

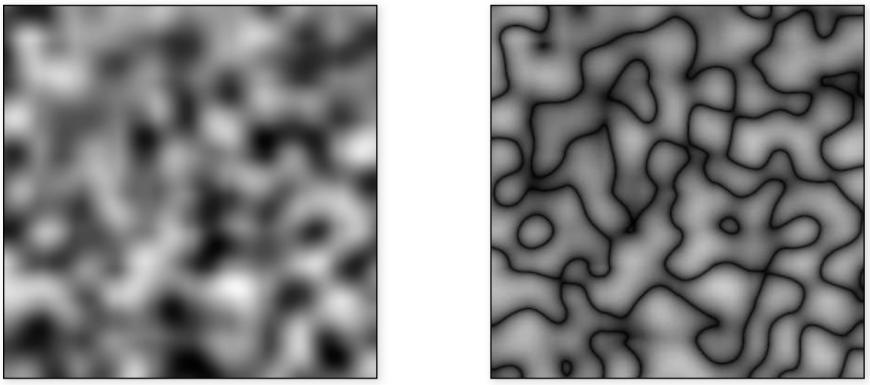


Figure 3: The left image illustrates 2D Perlin noise values ranging from $[0, 1]$, and the right image depicts 2D Perlin noise taking absolute values ranging from $[-1, 1]$ [28].

4.4 Environment design

The following section will explicate the different spaces in the Gymnasium environment, such as the action and observation spaces. Furthermore, it will explain important functions, such as score and transition. Numerous types of action spaces have been utilised in the experiments and will be described in this section.

4.4.1 Score function

The distance travelled by the agent determines the score in HCR; the same applies to the HCR environment. The score is calculated based on the following function:

$$score(x_{max}) = \max\left(0, \lfloor x_{max} - \texttt{X\_SPAWNING} \rfloor\right)$$

Where x_{max} is the maximum distance on the x-axis reached by the agent in a timestep of an episode and `X_SPAWNING` the x-axis spawning location of the agent. The function calculates the difference between x_{max} and `X_SPAWNING` rounded down to the nearest integer. The score can never be lower than 0 because of the max(0,·) function; it prevents negative scores that could occur if the agent moves backwards from its spawning location. When an agent reaches the maximum score (1000 or 300 in our experiments), the game (or episode) ends as it is completed.

4.4.2 Observation space

Initially, multiple distinct observation spaces were experimented with. The experiments revealed that the more observations, the better the agent performed, especially with algorithms such as PPO. Therefore, it was decided not to include those experiments in the thesis and to include all relevant parameters in the observation space in every experiment. A graph showing the learning curve of different observation space types can still be found in Appendix B.1. The observation space remains constant throughout all experiments.

The observation space in the environment is a composite space defined as a dictionary containing distinguishable fundamental spaces. HCR's observation space (see Appendix B.2 for code) contains four of these fundamental spaces, these are defined mathematically as described below:

1. *Car chassis position*

$$p = X \times Y, \quad \text{where} \quad X \in [0, 1000], \quad Y \in [0, 700] \quad \text{and} \quad X, Y \in \mathbb{R}$$

This observation indicates the car's position (p) based on the chassis and is defined as a Cartesian product where X is the x-coordinate and Y the y-coordinate of the car. Correspondingly, this fundamental space is defined as a 2-dimensional box space with a range limit from $[0, 1000]$ for the set X and $[0, 700]$ for set Y.

2. *Car chassis angle*

$$a = \theta, \quad \text{where} \quad 0 \leq \theta \leq 360 \quad \text{and} \quad \theta \in \mathbb{R}$$

The car's angle a is defined as the variable θ in a one-dimensional fundamental box space ranging from 0 to 360. The car's angle is likewise measured relative to its chassis. When the car completes a full rotation, the angle resets to 0 degrees.

3. *Wheels speed*

$$v = V_{back} \times V_{front}, \quad \text{where} \quad V_{back}, V_{front} \in [-13\pi - 0.1, 13\pi + 0.1] \quad \text{and} \quad V_{back}, V_{front} \in \mathbb{R}$$

Moreover, the wheel speed v is observed, which is the Cartesian product of the car's front (V_{front}) and back (V_{back}) wheels. The fundamental space for the wheel speed is similar to that of the car's position, though the range differs notably as it ranges from $[-13\pi - 0.1, 13\pi + 0.1]$ for both back and front wheel speeds. The motor speed determines the range, which is set to $13 \cdot \pi$. For mathematical compatibility purposes, 0.1 is subtracted and added as a margin of error.

4. *Wheels on the ground*

$$c = C_{back} \times C_{front}, \quad \text{where} \quad C_{back}, C_{front} \in \{0, 1\}$$

Lastly, the observation of whether the front or back wheel contacts the ground is denoted by c. A multi-binary space is utilised to observe both the front (C_{front}) and back (C_{back}) wheels, where 0 signifies not contacting the ground, and 1 means the wheel has contact with the ground.

These four different observations combined make the environment highly sophisticated, and thus, the curse of dimensionality will become evident when exploring it. The full observation space set O is defined as follows using the aforementioned definitions and variables.

$$O = (X \times Y) \times \theta \times (V_{back} \times V_{front}) \times (C_{back} \times C_{front})$$

Or the simplified version:

$$O = p \times a \times v \times c$$

4.4.3 Action space

As mentioned before, multiple action spaces were utilised in the experiments. Three action spaces are defined: discrete with three actions and continuous. Originally, a discrete action space with two actions was added. However, this action space was constantly outperformed by the discrete action space with three actions and hence was removed.

1. *Discrete action space with three actions*

$$Discrete(a) = \begin{cases} \text{Idle}, & \text{if } a = 0 \\ \text{Gas}, & \text{if } a = 1 \\ \text{Reverse}, & \text{if } a = 2 \end{cases}$$

   The first action space is a discrete action space with three actions. Consider a an action. Then these actions are idle when $a = 0$, gas when $a = 1$, and reverse when $a = 2$. This action space is the same one humans have when playing the game; in that case, idling would not be doing any action.

2. *Continuous action space*

$$Continuous(v) = v, \quad \text{where} \quad -13 \leq v \leq 13$$

   The continuous action space acquires a motor speed variable v. Afterwards, both front and back wheel speeds are adjusted to $v \cdot \pi$. The maximum motor speed is set to 13 in the environment, thus v ranges from -13 to 13. The code for these action spaces can be found in Appendix B.3.

4.5 Reward functions

The subsequent section will explicate how distinct reward functions are built up. The experiments will also utilise all of the following reward functions. Before explaining each reward function, one must understand the different predefined parameters used in these reward functions. The reward type parameters `soft` and `aggressive` define the significance of punishing specific actions such as idling or reversing. Consider that the reward type is a parameter and does not fully define a reward function.

The soft function $soft(c)$ with c as status is defined as follows:

$$soft(c) = \begin{cases} -0.1, & \text{if } c = \text{idle} \\ -0.2, & \text{if } c = \text{reverse} \end{cases}$$

And $aggressive(c)$ with c as status is defined as:

$$aggressive(c) = \begin{cases} -0.5, & \text{if } c = \text{idle} \\ -1, & \text{if } c = \text{reverse} \end{cases}$$

With $soft(c)$ and $aggressive(c)$ it is possible to define the reward type function $k(f, c)$, where f is the chosen reward type function:

$$k(f,c) = \begin{cases} soft(c), & \text{if } f = \text{soft} \\ aggressive(c), & \text{if } f = \text{aggressive} \end{cases}$$

Furthermore, each reward function is defined as $R(s, a, s', f, c)$, representing the reward obtained by applying action a in the current state s, leading to a transition to state s', the resulting state. The reward type is defined by f, and c represents the agent's status. Therefore, the reward r is discovered by calculating $r = R(s, a, s', f, c)$. However, a is not always included in the reward function calculation as this depends on the reward function.

4.5.1 Distance

The distance reward function utilises the agent's position observation on the x-axis in the current state and examines the maximum distance an agent has achieved prior (in the previous state). The function uses these observations to calculate the delta between these positions and utilises these differences to calculate the concluding rewards. This distance reward function $R_d(s, a, s', f, c)$ is defined in the following manner:

$$R_d(s,a,s',f,c) = \begin{cases} -100, & \text{if } s' \text{ is truncated} \\ -100, & \text{if } s' \text{ is terminated} \\ 0, & \text{if } s' \text{ is terminated by game completion} \\ k(f, reverse) + p_x(s') - p_{x_{max}}(s), & \text{if } p_x(s') < p_{x_{max}}(s) \\ k(f, idle), & \text{if } \big(p_x(s') - p_{x_{max}}(s)\big) < 0.001 \\ 1 + p_x(s') - p_{x_{max}}(s) & \text{if } p_x(s') > p_{x_{max}}(s) \end{cases}$$

Where $p_x(s')$ is the (current) x-coordinate position of the agent in state s' and $p_{x_{max}}(s)$ the (previous) maximum x-coordinate position in state s. And $k(f, c)$ the aforementioned reward type function where $c = idle$ or $c = reverse$. It's worth noting that the failure of the agent to advance from its previous state results in a negative reward. Similarly, moving forward but covering a distance of less than 0.001 also leads to a negative reward. A positive reward of is awarded only when the agent successfully moves forward from its previous state.

4.5.2 Action

The action-based reward function utilises the agent's action to compute a reward. Depending on the action, the agent receives a reward bound to this action. The reward function $R_a(s, a, s', f, c)$ is stated as below:

$$R_a(s,a,s',f,c) = \begin{cases} -100, & \text{if } s' \text{ is truncated} \\ -100, & \text{if } s' \text{ is terminated} \\ 0, & \text{if } s' \text{ is terminated by game completion} \\ k(f, idle), & \text{if } a = 0 \quad \text{(idle)} \\ 1, & \text{if } a = 1 \quad \text{(gas)} \\ k(f, reverse), & \text{if } a = 2 \quad \text{(reverse)} \end{cases}$$

Where a is the chosen action in state s and $k(f, c)$ is the reward type function. A reward of 1 is given when the agent decides to choose the gas action. Meanwhile, a negative reward is given when the agent is idling: -0.5 when $f = aggressive$ and -0.1 when $f = soft$. Likewise, when the chosen action is reverse, a harsher negative reward is given: -1 when $f = aggressive$ and -0.2 when $f = soft$.

4.5.3 Wheel speed

Comparable to the previous reward functions, an observation is made to see whether the agent is moving forward, moving in reverse, or idling. However, the wheel speed reward function relies on the wheel speed rather than the actions or distance observations. The wheel speed reward function $R_w(s, a, s', f, c)$ is defined as described below:

$$R_w(s, a, s', f, c) = \begin{cases} -100, & \text{if } s' \text{ is truncated} \\ -100, & \text{if } s' \text{ is terminated} \\ 0, & \text{if } s' \text{ is terminated by game completion} \\ k(f, idle), & \text{if } -1 \leq v_{back}(s') \leq 1 \text{ and } -1 \leq v_{front}(s') \leq 1 \\ 1, & \text{if } v_{back}(s') < 0 \text{ and } v_{front}(s') < 0 \\ k(f, reverse), & \text{if } v_{back}(s') > 0 \text{ and } v_{front}(s') > 0 \end{cases}$$

Where $v_{back}(s')$ is the speed of the back wheel and $v_{front}(s')$ is the speed of the front wheel. Again the function $k(f, c)$ is utilised as defined before. Note that $c = idle$ when the wheel speed is close to 0, specifically in the range $[-1, 1]$. Additionally, note that the wheel speed is negative when the agent moves forward. Accordingly, a reward of 1 is assigned when the wheel speed is below 0 and a negative reward when the wheel speed is above 0.

4.5.4 Airtime

The airtime reward is an additional reward function that extends the distance and wheel speed functions. It attempts to increase the agent's airtime by positively rewarding multiple consecutive timesteps in the air. Meanwhile, the reward functions of the distance and the wheel speed are unchanged. For every 30 consecutive frames in the air, a reward of 1 is allocated until the agent contacts the ground again; this airtime function is defined as $A(s) = s_{t_{air}}/30$, which entails the airtime counter in state s and where $s_{t_{air}}$ is the number of timesteps spent in the air consecutively in s. For each timestep in which the agent contacts the ground, the agent is punished with a negative reward of -0.5. The airtime reward function is defined as follows:

$$R_{air_i} = R_i(s, a, s', f, c) + R_{air}(s, a, s')$$

Where $R_i(s, a, s', f, c)$ is one of the aforementioned reward functions (wheel speed or distance) and $R_{air}(s, a, s')$ is defined as:

$$R_{air}(s, a, s') = \begin{cases} A(s'), & \text{if } A(s') > 0 \\ -0.5, & \text{otherwise} \end{cases}$$

Where $A(s')$ is the airtime number at state s'. For example, when the agent has spent 60 consecutive frames ($t_{air} = 60$) in the air, the agent receives an airtime reward of $R_{air}(s, a, s') = A(s') = 60/30 =$

2. This airtime reward R_{air} is then added to the reward of the other reward function R_i, which can represent the wheel speed or distance reward function. Thus the definitive formula concludes to: $R_{air_i} = R_i(s, a, s', f, c) + R_{air}(s, a, s')$.

4.6 Algorithms

This section will explore the algorithms used in the experiments. Specifically, the focus is on proximal policy optimisation (PPO). However, the random agent algorithm is used as a control benchmark for the experiments.

4.6.1 Random agent

The random agent is a policy that chooses the action uniformly at random. It utilises the action-based reward function and the discrete action space with three actions (idle, gas and reverse). For a random agent with a discrete action space of size $|A| = 3$, the policy is:

$$\pi(a \mid s) = \frac{1}{|A|} = \frac{1}{3} \quad \forall a \in A, s \in S$$

Where $\pi(a \mid s)$ is the probability of taking action $a \in A$ given state s in state space S. The full algorithm can be found in Algorithm 1.

Algorithm 1 Random Agent

Initialisation: current timestep: $t = 0$, maximum number of timesteps: T
Run Episodes:
for each episode i **do**
    Reset environment to get initial state s_0.
    while termination condition (terminated or truncated) is not met and $t < T$ **do**
        Select action a_t according to policy $\pi(a_t \mid s_t)$: $a_t \leftarrow \text{Uniform}(A)$
        Execute action a_t in the environment.
        Observe next state s_{t+1}, reward r_t, and termination signal.
        Update current timestep: $t \leftarrow t + 1$
    end while
end for

4.6.2 Proximal Policy Optimisation (PPO)

PPO is a policy gradient algorithm that utilises a combination of trust region methods (originally TRPO), actor-critic methods, policy gradient methods and a clipped surrogate objective function to solve reinforcement learning problems. It is seen as a state-of-the-art reinforcement learning algorithm on continuous control tasks because it is data efficient, scalable to large models, performs relatively better compared to other algorithms and is robust as it is successfully applied to numerous problems without extensive hyperparameter optimisation [29]. Moreover, PPO utilises an actor-critic network where the actor (policy network) controls how the agent behaves by outputting a probability distribution over actions given a state s_t at timestep t, while the critic (value network)

evaluates a state s_t and outputs the expected cumulative reward for s_t under the current policy. The critic and policy networks are implemented as feed-forward neural networks.

Before explicating PPO further, it is essential to know why PPO originated. Firstly, policy gradient methods such as REINFORCE [30] are often too aggressive with the modifications in the policy updates. More minor and conservative policy updates increase the chance of converging to an optimal solution instead of having extensive policy updates that result in bad policies. PPO successfully attempts to improve upon these gradient policy methods by introducing a clipped surrogate objective function. This function constrains policy updates between the new and old policy to lie within a specified range. This clipping mechanism helps to stabilise training and improve the reliability of policy updates as they stay closer to the old policy.

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

Where θ defines the parameters of the policy network, the ratio reward function $r_t(\theta)$ calculates the probability ratio between the current policy and the old policy, $r_t(\theta)$ is defined as stated below:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

$\pi_\theta(a_t \mid s_t)$ defines the probability of taking an action a_t in state s_t and $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ does the same for the previous (old) policy. $\hat{A}_t$ is the advantage estimator, which measures the relative quality of a specific action compared to the average action in a given state s_t. Specifically, the General Advantage Estimator (GAE) is utilised, which combines the ideas of temporal difference learning and Monte Carlo estimation to compute the advantage [31]. ϵ a hyperparameter that defines the clip range, $\epsilon = 0.2$ is used in the experiments (as is originally in the paper). Hence, the policy update will always be in the scope of the clipping range, which is $[1-\epsilon, 1+\epsilon]$.

Moreover, a value loss function is defined using the Mean Squared Error (MSE) between the predicted estimated cumulative rewards in a state s_t and the observed cumulative rewards at t. The value loss function $L^{\text{VF}}(\phi)$ utilised in PPO is stated below:

$$L^{\text{VF}}(\phi) = (V_\phi(s_t) - V_t^{targ})^2$$

Where V_t^{targ} is the observed cumulative reward, $V_\phi(s_t)$ is the value predicted by the value (critic) network for state s_t, and ϕ are the parameters of the value network. MSE is a function that calculates the mean-squared error. Lastly, there is an entropy bonus function $S[\pi_\theta](s_t)$ that is added to the objective function to regulate exploration; it calculates the entropy of policy $[\pi_\theta]$ in-state s_t. Using all the aforementioned defined functions, the full objective function for PPO is defined as follows:

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

Where c_1 and $c2$ are coefficients that scale the significance of the value function loss and entropy bonus relative to the clipped surrogate objective. The pseudocode can be found in Algorithm 2. The PPO algorithm utilised in the experiments all originate from the Stable-Baselines3 library [32].

Algorithm 2 PPO from Stable-Baselines3 [32]

Initialisation: policy parameters: θ, value function parameters: ϕ
for iteration = 1, 2, ... **do**
  for actor = 1, 2, ... **do**
    Run policy $\pi_{\theta_{old}}$ in the environment for t timesteps to collect a set of trajectories .
    Compute advantage estimates $\hat{A}_t$ using GAE
  end for
  Update the policy by maximizing the PPO objective function $L^{CLIP+VF+S}(\theta)$
  Update the value network parameters ϕ and minimise the value loss function $L^{\mathrm{VF}}(\phi)$
  $\theta_{old} \leftarrow \theta$
end for

5 Experiment design & results

The following experiments are conducted to construct an agent capable of completing the HCR environment with the most promising potential performance. This section explores the reward design, determining what reward functions can help the agent perform the best. We also analyse the performance of agents utilising the discrete and continuous action space. Moreover, the best-performing agents are evaluated and observed in an environment with a distinguishable difficulty mechanic (algorithm) for terrain generation. Further, an experiment is conducted to test if a specific reward function can incentivise the agent to achieve more airtime.

5.1 Random agent

An experiment was executed with a random agent in a discrete action space with three actions (idle, gas, and reverse) using both the soft and aggressive action-based reward functions, as shown in Figure 4. The chosen maximum score in this experiment is 1000. The experiment advances this random agent through 1 million timesteps in the HCR environment, and the action is selected randomly using the uniform distribution. This experiment was run five times, and the results were averaged. The motivation for this experiment is to examine whether other experiments perform better than a random agent that does not learn.

These graphs clearly show that the random agent's performance is low across all metrics, as the learning curve, score, and episode length curve are all flat. The score is, on average, 15, and the average rewards do not reach higher levels than 100. This is expected behaviour for a random agent, as it does not learn or adapt its strategy based on the environment or past experiences. The average reward curve distinctly shows the difference between the reward type (soft and aggressive) functions, where the punishment is more harsh using the aggressive reward function, as the reward curve seems to stay around -200. In contrast, the soft reward curve stays above 0. These results can be utilised as a baseline comparison for upcoming experiments to determine if the agent learns and improves its performance. The episode length and score do not differ extensively, as the reward type curves are roughly the same.

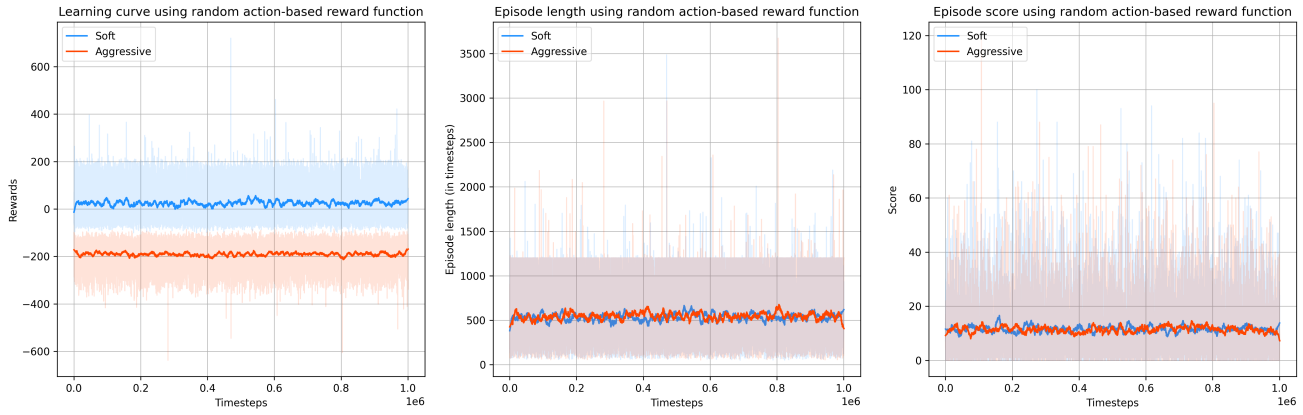


Figure 4: Graphs illustrating the performance of the random agent experiment during the training phase: the rewards learning curve (left), episode length (middle), and episode score (right) over 1 million timesteps. Results are averaged over 5 runs, with the shaded area indicating the variability across these runs.

5.2 Discrete action space

Numerous experiments employing different reward functions have been performed using the PPO algorithm in a discrete action space. The experiment involves training an agent using different reward functions: action-based, distance-based, and wheel speed-based. Each experiment was executed five times, with the results curve averaged, and the variability over all five runs is shaded in the graphs. Each experiment ran for 1 million timesteps. The maximum achievable scores were set to 300 and 1000 in separate experiments, and these results were compared.

5.2.1 Action-based

The action-based reward functions experiment demonstrates weak results, as shown in Figure 5. In particular, the aggressive reward function does learn and performs worse than the random agent on average. However, the soft reward function does display some learning in the first 100,000 timesteps, where it outperforms the random agent with average rewards above 100 and average scores of almost 50. Nevertheless, there is some performance degradation after timestep 100,000, probably due to overfitting on the beginning difficulty of the level. Because the terrain always starts relatively flat, it might be difficult for the agent to learn the dynamicity of the environment as the terrain accumulates height. Interestingly, there are many episodes where the agent finishes the complete environment by reaching a score of 300, though it performs poorly on average.

In the same experiment, but with a higher maximum score of 1000 in the environment, the soft action-based reward function performed significantly better. The agent achieves average rewards of around 1200 and average scores above 400, as seen in Figure 6. One possible explanation for this performance increase is that the environment with a maximum score of 1000 starts with relatively flatter terrain for a longer duration. In contrast, the terrain in the 300 maximum score environment becomes significantly steeper more quickly. This is due to linear interpolation in terrain generation and the identical peak difficulty in both environments. Additionally, the agent only

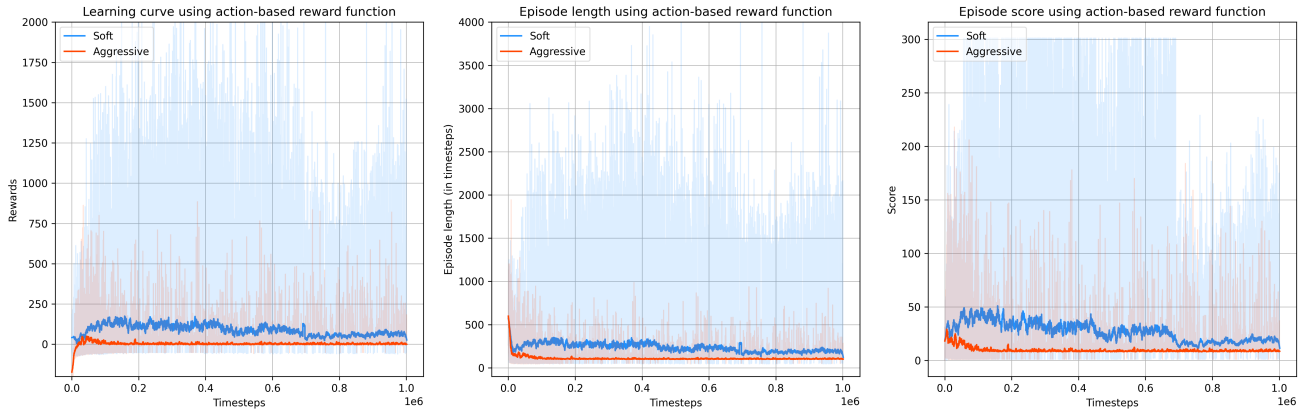


Figure 5: Graphs illustrating the performance of the discrete action-based reward function experiment in an environment with a maximum achievable score of 300 during the training phase.

seems to complete the entire environment four times by reaching a score of 1000, while the same agent completes the environment more frequently in the 300 environment. However, the aggressive reward function still performs poorly, likely caused by the harsh negative rewards when the agent is idling or moving backwards, which results in an inadequate policy.

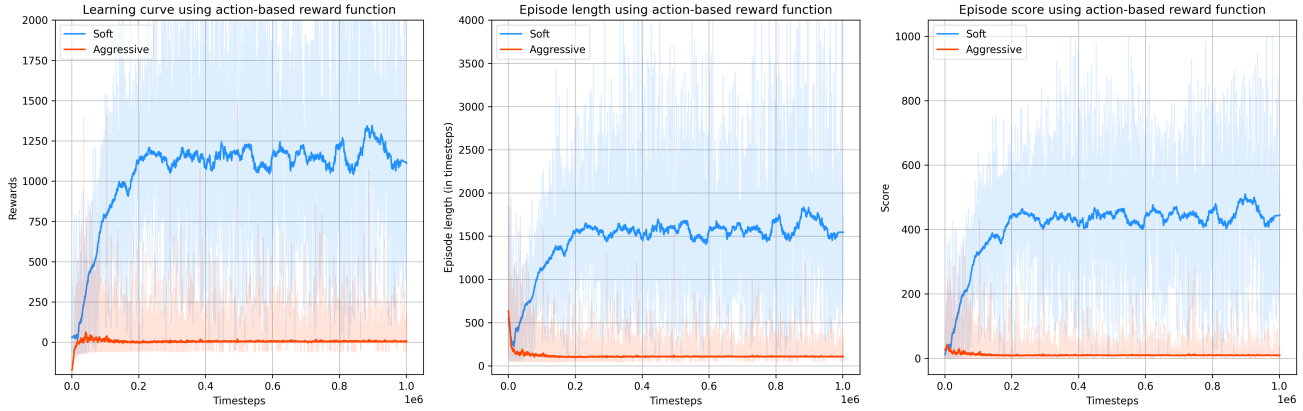


Figure 6: Graphs illustrating the performance of the discrete action-based reward function experiment in an environment with a maximum achievable score of 1000 during the training phase.

5.2.2 Distance-based

Figure 7 shows the distance-based reward function experiment graph. The distance-based reward function in the 300 environment performs better than the action-based reward function. Both reward type functions perform similarly and achieve scores of around 200, and both achieve completing the environment multiple times. The severe negative rewards in the aggressive reward function seem to impact the agent less compared to the action-based environment. Although there are extreme negative reward spikes in the learning curve. These are induced when the agent is stuck on the terrain, indicating it no longer traverses the terrain; the negative idle rewards stack up to 1200

timesteps, and then the environment calls for truncation.

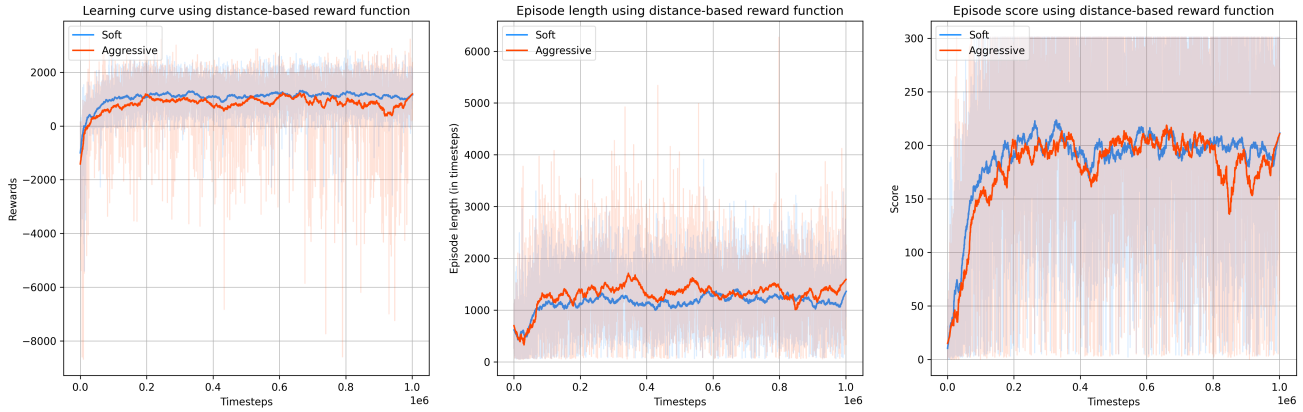


Figure 7: Graphs illustrating the performance of the discrete distance-based reward function experiment in an environment with a maximum achievable score of 300 during the training phase.

The 1000 environment in Figure 8 demonstrates the aggressive reward function to be slightly worse than the soft reward function regarding episode score. Again, extreme negative reward spikes can be found in the learning curve. Both environments are able to complete the environment several times. The training reward curve converges in both environments, at 1500 for the 300 environment and around 2000 for the other environment. The 300 environment appears to perform better because it traverses further in the environment when looking at the average percentage of level traversed. Also, the 300 maximum score environment completed the environment way more often than the 1000 environment.

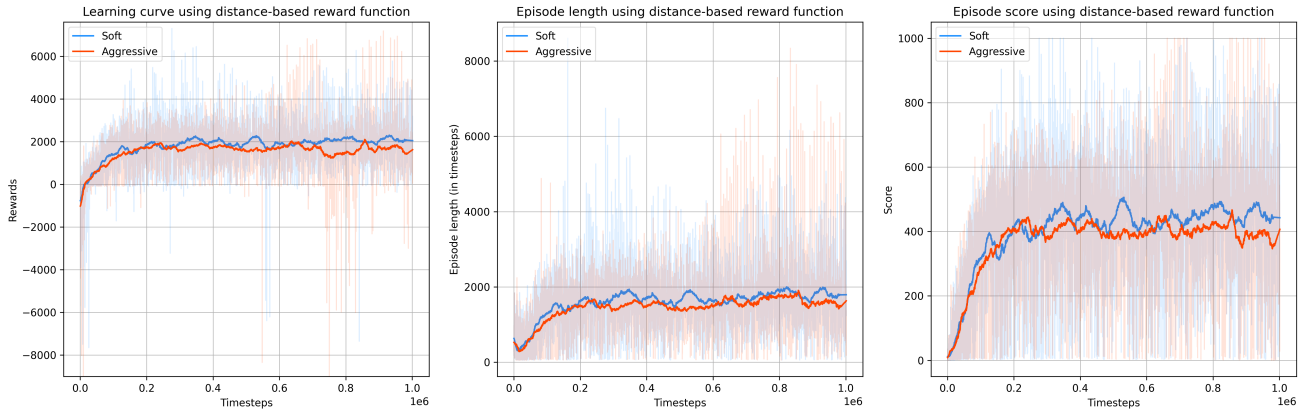


Figure 8: Graphs illustrating the performance of the discrete distance-based reward function experiment in an environment with a maximum achievable score of 300 during the training phase.

5.2.3 Wheel speed-based

Moreover, we utilised a wheel speed-based reward function and PPO to train an agent. This experiment with a maximum score of 300 displayed interesting results in Figure 9; both reward type agents seem to learn and perform better than the random agent, though the performance is inferior to the previous distance-based reward function experiment. The aggressive reward function here clearly outperforms the soft reward function. The reason for this inadequate performance could be that the agent drives deliberately slowly (just slow enough to prevent truncation) to maximise the wheel speed rewards until the episode ends. Slow driving would also negatively interfere with the learning policy in PPO as the policy observes positive rewards and thus promotes slow driving. Looking at the score, the agent performs worse after peaking at around score 200, which is around 390.000 timesteps. The slow driving becomes even more apparent when comparing the episode time of this agent to the distance-based reward function agent. The distance-based agent seems to complete a score of 200 at around 1000 timesteps, while the wheel-speed-based agent needs more than 2000 timesteps for the same score.

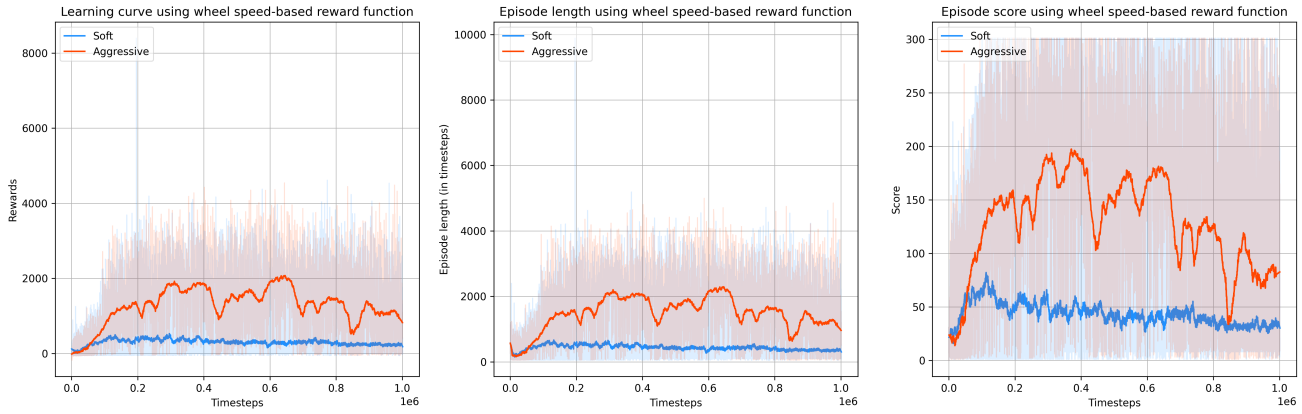


Figure 9: Graphs illustrating the performance of the wheel speed-based reward function in a discrete action space experiment with a maximum achievable score of 300 during the training phase.

The agent's performance in the 1000 maximum score environment, as depicted in Figure 10, is notably poor. This graph vividly illustrates the agent's slow driving behaviour. In episodes where the agent reaches the 1000 maximum score, the episode length extends to nearly 14,000 timesteps, twice as long as the distance-based reward function agent. Both reward type functions perform poorly and do not seem to improve the policy to perform better in the environment at all. Only in the first 100,000 timesteps does the agent seem to learn in such a way that results in a policy with better game scores. Afterwards, it might increase the reward's learning curve, but the game score does not improve significantly.

5.2.4 Score comparison

Figure 11 demonstrates box plots with the episode scores of each aforementioned reward function experiment while training across all runs of 1 million timesteps. In the 300 score environment, Figure 11a clearly shows that the distance reward function has performed the best, with average scores

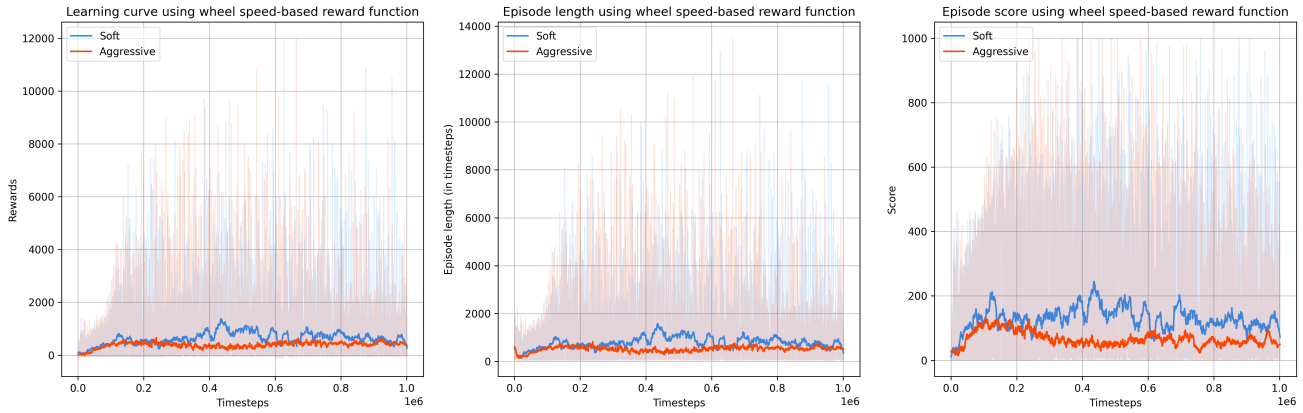
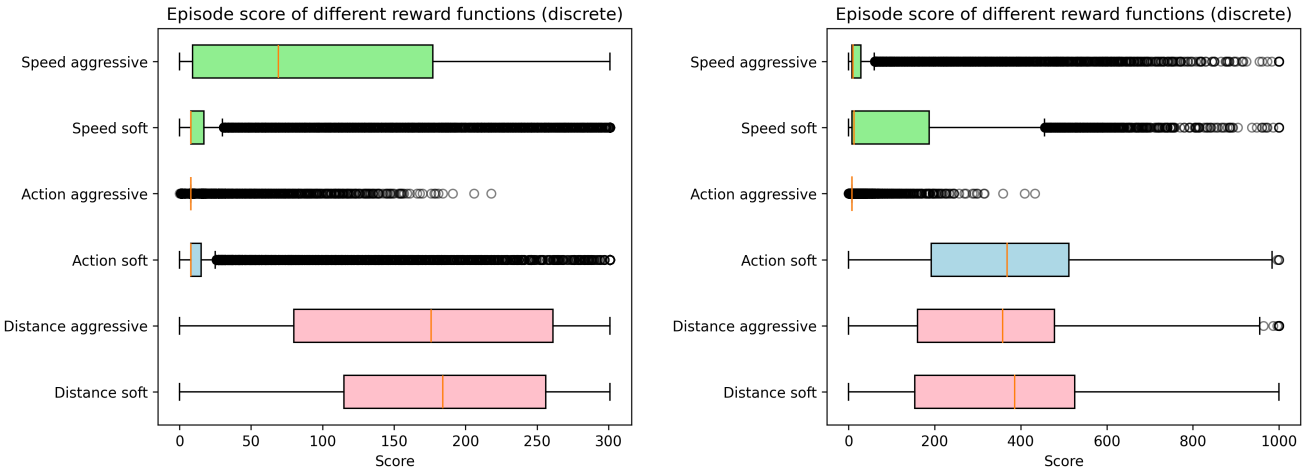


Figure 10: Graphs illustrating the performance of the discrete action space and wheel speed-based speed reward function experiment in an environment with a maximum achievable score of 1000 during the training phase.

above 175 for both soft and aggressive reward functions. The soft wheel speed and action-based reward functions have a low median score of under 25. Nonetheless, one can observe episodes where the agent completed the environment multiple times from the outliers, except for the agents in the aggressive action-based reward experiment. The 1000 score environment shows similar results (see Figure 11b), yet the aggressive wheel speed reward function performs poorly, with a median score below 10. Additionally, the soft action-based reward agent performs reasonably, almost identically to the distance-based reward agents. In the discrete action space environment, it seems that the distance-based reward agents perform the best, followed by the action-based agents and, consequently, the wheel-speed agents.



(a) Maximum score of 300

(b) Maximum score of 1000

Figure 11: Box plots illustrating the episode scores of different reward functions across all training runs in the HCR environment in two different maximum score experiments. Speed represents the wheel speed reward function, and the orange line in the bar represents the median.

5.3 Continuous action space

Analogous to the previous experiments in a discrete environment, we conducted a series of experiments in a continuous action space utilising the PPO algorithm. These experiments again involved training an agent with various reward functions, specifically wheel speed-based and distance-based. Each training session was repeated five times to ensure the reliability of our results. The resulting curves were then averaged, with shaded areas representing the variability across all five runs. The duration of each experiment is 1 million timesteps. Again, the experiments have been conducted for a maximum score of 300 and 1000. Regardless, only in the score comparison (Section 5.3.3) do we show the results of the 300 score environment because the training curves were virtually identical between the 1000 and 300 environments. The results of the 300 score environment can still be found in Appendix C.1.

5.3.1 Distance-based

The distance-based reward function with the continuous The distance-based reward function with the continuous action space experiment demonstrates promising results (see Figure 12, as the average score reached almost 700 after 200,000 timesteps. However, after the 200,000th timestep, the policy deteriorates in both the rewards learning curve and the episode score until around the 400,000th timestep; after that, it seems to converge to 3000 rewards. It appears PPO overfits the policy when the agent reaches more elevated terrain. The overfitting causes the agent to fail to drive to the episode's earlier stages, resulting in lower rewards and episode scores. Additionally, the continuous action space utilises an extensive range of numbers, increasing the problem's dimensionality. Furthermore, the episode lengths seem rather lengthy compared to other experiments. The two reward type functions do not differ much; the soft reward function seems to achieve higher scores and overall higher rewards.

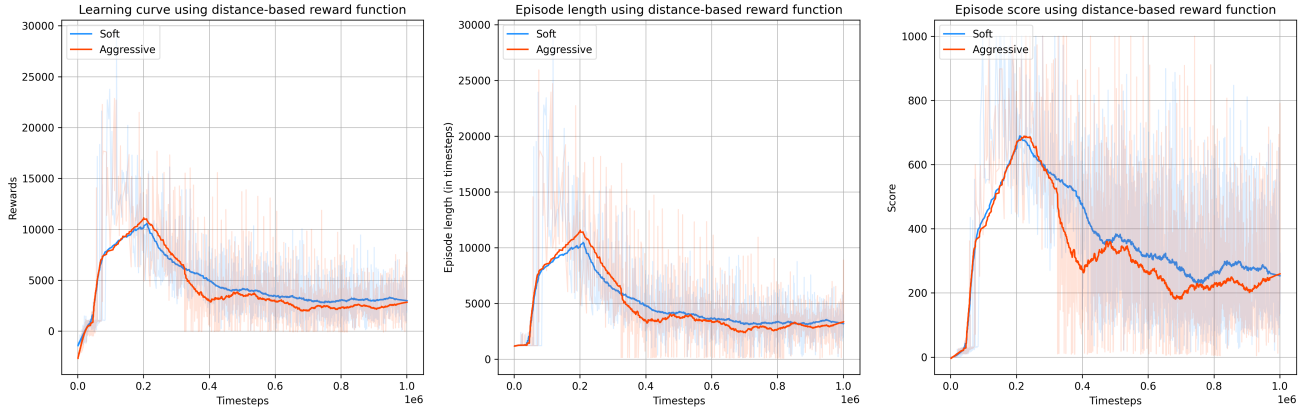


Figure 12: Graphs illustrating the performance of the distance-based reward function and continuous action space experiment in an environment with a maximum achievable score of 1000 during the training phase.

5.3.2 Wheel speed-based

Whereas the distance-based reward function has difficulty converging, the wheel speed-based reward function has no trouble accomplishing this in the continuous action space experiment. In Figure 13 one can observe the reward learning curve to converge after the 400,000th timestep to approximately 13,000 in rewards. The score learning curve converges at around 800, making this specific reward function and action space combination the best-performing one yet. The reasoning for PPO not achieving the 1000 score might be that the terrain only increases its difficulty until 80% (meaning 800 scores) is reached. Afterwards, the terrain height difficulty stays the same, and this could initiate a difficulty for PPO as a significant adjustment in the policy is needed. This hypothesis will be tested in Section 5.5. Nevertheless, the episode lengths are again lengthy for the wheel speed-based reward function; the agent's speed is further investigated in Section 5.4.2. Consider again that the continuous action space utilises the wheel speed as an action to traverse the HCR environment. The wheels-speed-based function performs superiorly for this specific action space as presumably both the reward function and action space utilise the same variable, leading to faster convergence and proper learning and increasing the episode score. Moreover, there seems to be no significant difference between the soft and aggressive reward functions, and these perform comparably to each other.

Figure 13: Graphs illustrating the performance of the wheel speed-based reward function and continuous action space experiment in an environment with a maximum achievable score of 1000 during the training phase.

5.3.3 Score comparison

As seen in the aforementioned sections, the wheel speed-based reward function performs the best, with little distinction between reward type functions; aggressive might barely outperform soft. It achieves a median score of around 700 in the training phase, while the distance-based reward function only achieves a score of around 190. Here, the soft distance-based reward function achieves higher scores than the aggressive distance-based reward function, although only faintly. Furthermore, the episode score range of the continuous action space experiments seems similar, as the whiskers and boxes are around the same size and length. The distance-based reward function performed better during training when utilising the discrete action space, as it had a median score of almost

400 compared to only 190 in the continuous action space. On the other hand, the wheel-speed-based reward function performed much better in the continuous action space; the agent could not even reach a median of 100 in the discrete action space. There appears to be a relation between specific reward functions and action spaces.

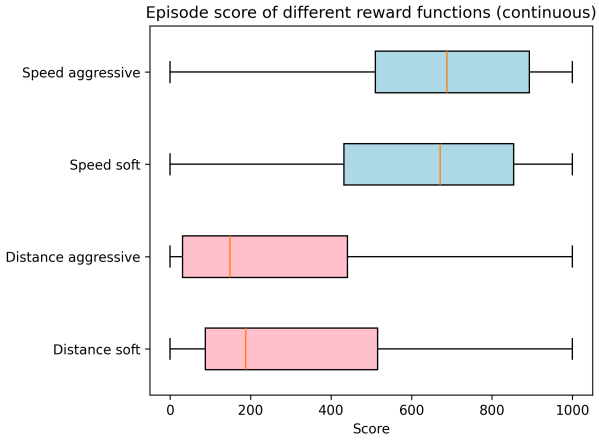


Figure 14: Box plot illustrating the episode scores of a PPO agent in the training phase utilising the distance and wheel speed reward functions in the HCR environment. The experiment utilises the continuous action space and a 1000 maximum score. Speed represents the wheel speed reward function, and the orange line in the bar represents the median.

5.4 Evaluation of best models

5.4.1 Score

The subsequent experiment evaluated the best models from the previous experiments. Each evaluation examines 1000 episodes, and PPO is again utilised. We will evaluate the distance-based and action-based soft reward functions in the discrete action space selecting actions stochastically as this performed better. Moreover, both reward type wheel speed-based reward functions and the soft distance-based reward function models in the continuous action space were evaluated using deterministic action selection. These specific combinations of a reward function and action space were chosen as they performed the best in the prior experiments. Multiple models for each reward function were present in the five experiment runs. Hence, we cherry-picked the best models and evaluated them. Figure 15 reveals that the wheel speed-based reward function in the continuous action space performs the best, with a median score of around 750. Moreover, there is barely any difference between the soft and aggressive reward functions, with aggressive portraying slightly higher scores. The distance-based reward function in the discrete action space follows with a median score of almost 600, and lastly, the action-based reward function with a median score of 400. Moreover, Table 1 displays the mean score of the different evaluated reward functions with similar results to the median values in the box plot.

Experiment name	Score	Length (timesteps)	Speed (score/timestep)
Distance discrete soft	574	2299	0.250
Distance continuous soft	528	4833	0.109
Wheel speed continuous aggressive	773	13185	0.059
Wheel speed continuous soft	765	13316	0.057
Action discrete soft	396	1349	0.294

Table 1: Table displaying the mean episode score, mean episode length and the speed of different agents after 1000 episodes evaluation in the HCR environment.

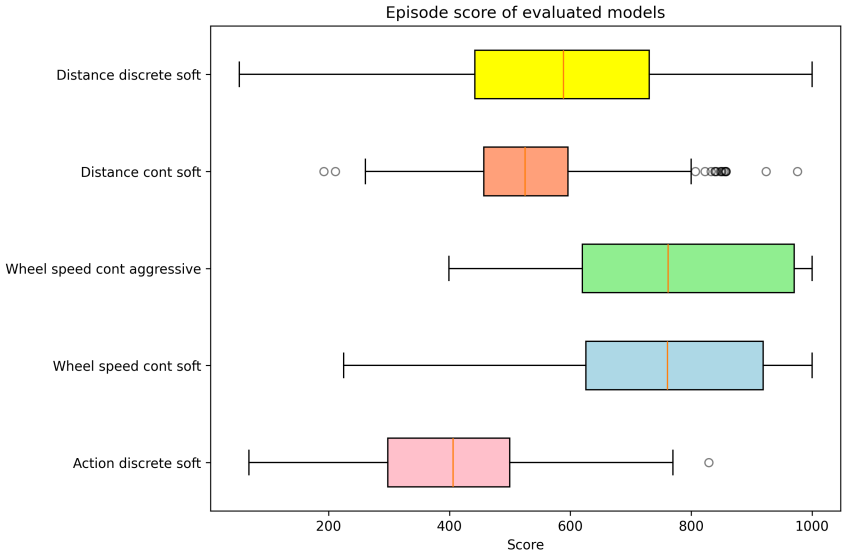


Figure 15: This box plot represents the episode scores of different models after evaluation of 1000 episodes. The orange line represents the median, and 'cont' denotes the agent's action space to be continuous.

5.4.2 Agent's speed

As noticed in the previous experiments, the wheel speed-based agents were examined to be relatively slow. This slow driving is confirmed by the average speed variable in Table 1, which is calculated by dividing the mean score by the mean length of an episode. The action-based agent in the discrete action space environment is almost five times faster than the wheel speed-based agents. Likewise, Figure 16 reveals the average agent's position (y-axis) in the HCR environment during an episode with timesteps on the x-axis. The same graph demonstrates agents utilising the discrete action space are the fastest, with nearly straight linear lines running up. Only 5100 timesteps are needed to complete the environment for the distance-based discrete action space agent, while the wheel speed-based agents need 17500 timesteps. The distance-based agent in continuous action space finishes second in terms of speed, as it requires around 9000 timesteps on average to complete the entire level.

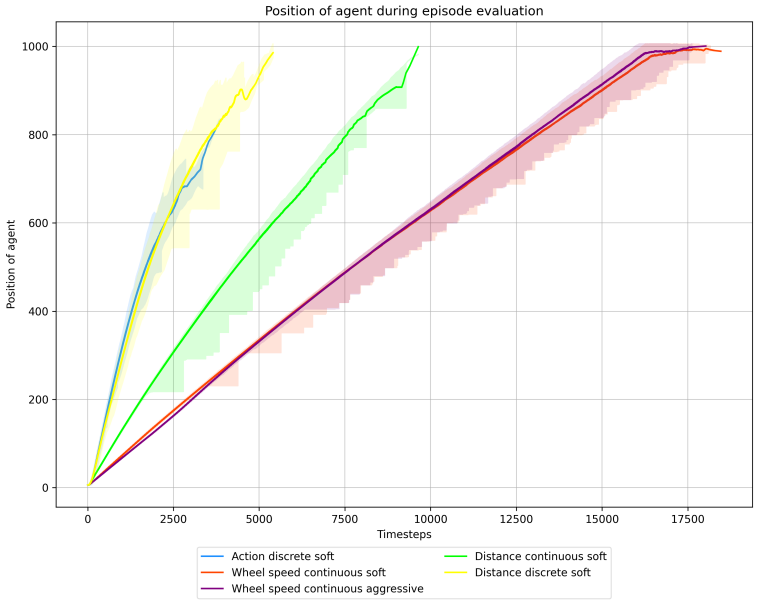


Figure 16: Line graph representing the agent's position in an episode after evaluation of 1000 episodes. The x-axis represents the duration of an episode in timesteps

5.5 Difficulty increase

Section 5.3.2 discusses the potential flaw of the agent not completing the environment, which is the difficulty not increasing until the end of the terrain. Instead, the difficulty only increases until 80% of the terrain with no modifications after 80%. This could be the explanation for the agent not converging to a maximum score of 1000. Hence, an experiment is conducted where the terrain difficulty increases until the end of the level. The experiment was conducted with all reward functions and action space combinations. However, only the best reward type was experimented with as those are primarily relevant. Specifically, action-based soft in a discrete environment, distance-based soft in continuous and discrete environments, and wheel speed-based aggressive in a continuous action space environment. Figure 17 illustrates these reward functions in the new environment in the training phase. The identical reward functions graph for the original difficulty (implying the terrain difficulty increases until 80% of the level) environment can be encountered in Appendix C.2 for comparison. Results show that all reward functions perform better than the original regarding episode scores except for the action-based reward function. The wheel speed-based reward function in continuous action space achieves a score of 1000 consistently after 200,000 timesteps. It is the best-performing agent yet, the original difficulty increase mechanic seems to be a problem for PPO to learn the optimal policy. In spite of that, the reward curve appears to decrease in average rewards after 200,000 timesteps because of likely overfitting. Similarly, distance-based reward functions achieved higher scores than originally, but the reward curves did not seem optimal. Both learning curves display a significant decrease in rewards and inconsistent scores. The same outcomes are found in Figure 18, which shows an episode score box plot of the same experiment compared to the original. Additionally, the action-based reward function in the new experiment performs inadequately and is not comparable to the original; it seems this particular agent has trouble learning consistently because it has many episodes where it completes the environment but

an inferior median score.

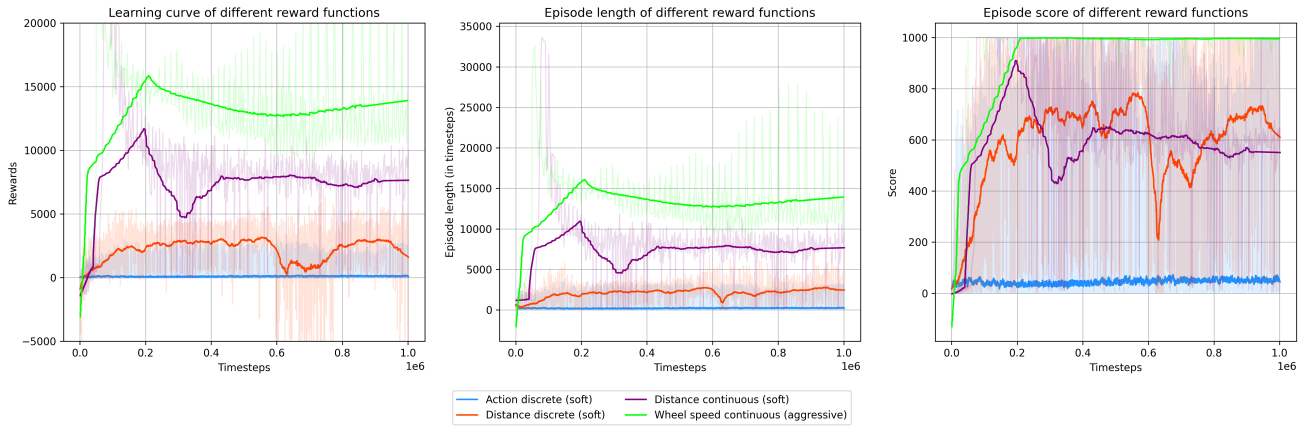


Figure 17: Graph representing the reward learning curve, episode length and score of different agents being trained in an environment where the terrain difficulty increases until the end.

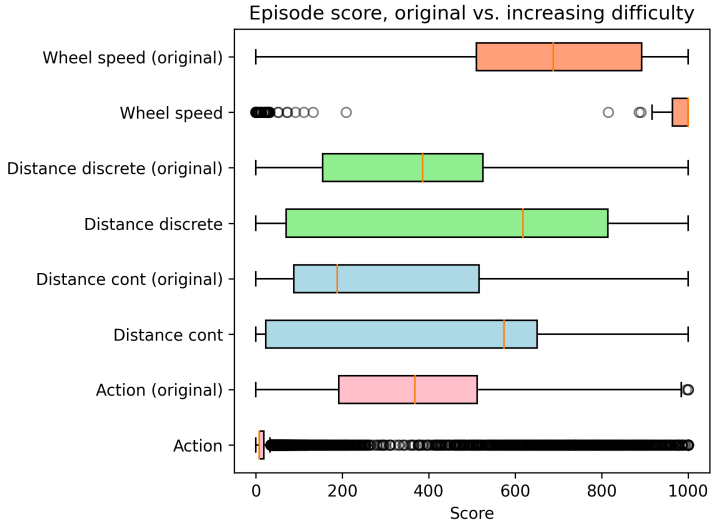


Figure 18: This box plot portrays the episode scores during the training of different agents in an environment where the terrain difficulty increases until the end compared to the original difficulty. The orange line represents the median, and 'cont' denotes the agent's action space to be continuous.

5.6 Airtime

Different airtime reward functions are defined to analyse whether increasing the agent's airtime is achievable and perhaps the airtime incentive raises the episode score. The experiment was executed with the distance-based and wheel speed-based reward functions in the continuous action space. Additionally, the distance-based reward function in the discrete action space is utilised. Similar to the previous experiments, the agent is trained in the HCR environment with one million timesteps

across five distinct runs. An agent's total airtime $A(s)$ (defined in Section 4.5.4) is observed, including the episode score, length and reward curves, which are uncovered in Figure 19. The same experiment has been executed utilising the original reward functions, including the airtime counter, which can be found in Appendix C.3.

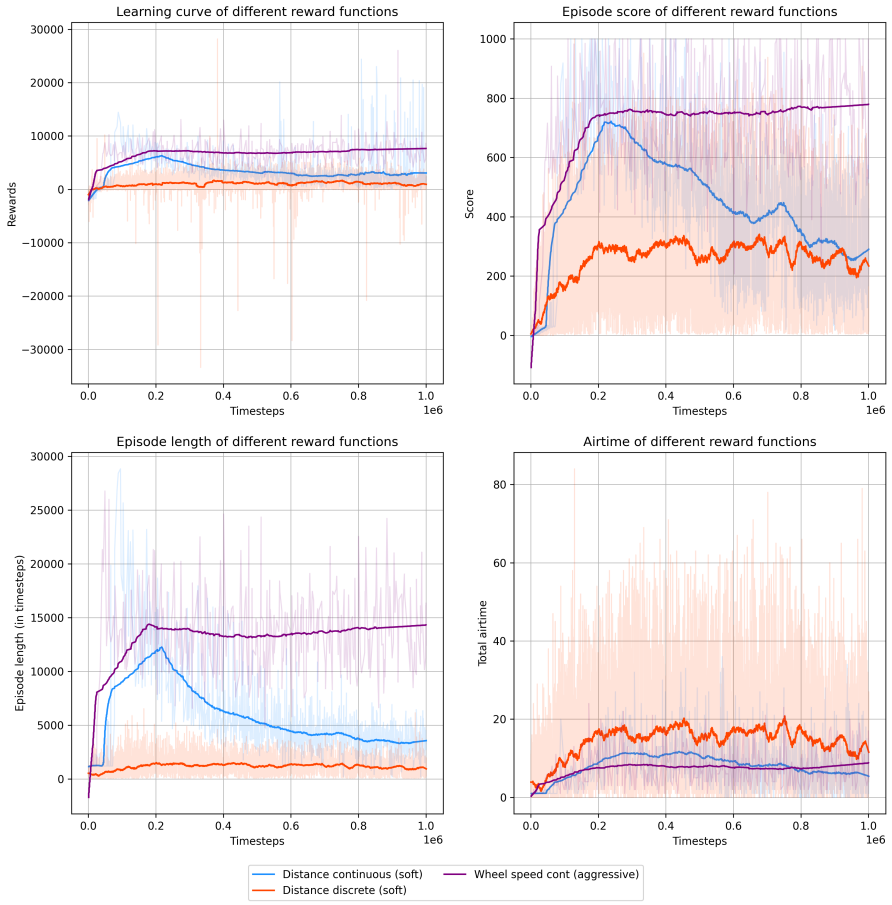


Figure 19: Graph representing the reward learning curve, episode length, score and total airtime of different agents being trained in the environment utilising the airtime reward functions.

As seen in the reward curve, the airtime function allocates less reward because of the -0.5 penalty in reward when the agent contacts the ground. This penalty also negatively influences the episode score, resulting in lower scores. The boxplots in Figure 20 reveal the episode score and total airtime from the original and airtime experiments. The score box plot likewise indicates lower median scores, except for the distance-based reward function in the continuous environment, which does show score improvement. There seems to be no significant increase in airtime, yet the airtime does increase with higher scores as the agent has more frames to perform airtime. The length/airtime (timesteps needed for one airtime count) observation in table 2 illustrates that more timesteps are required to achieve one airtime count for all airtime reward functions compared to their original. The airtime reward function does not promote airtime at all.

Experiment name	Score	Length	Airtime	Length/airtime
Airtime distance discrete	330	1318	16.6	79
Original distance discrete	341	1398	18.7	75
Airtime distance continuous	291	4047	5.5	730
Original distance continuous	127	1865	2.7	688
Airtime wheel speed continuous	620	10528	6.8	1548
Original wheel speed continuous	627	10583	8.5	1248

Table 2: Table displaying the mean episode score, mean episode length (in timesteps), mean airtime count and the required timesteps for one airtime count of different agents during training.

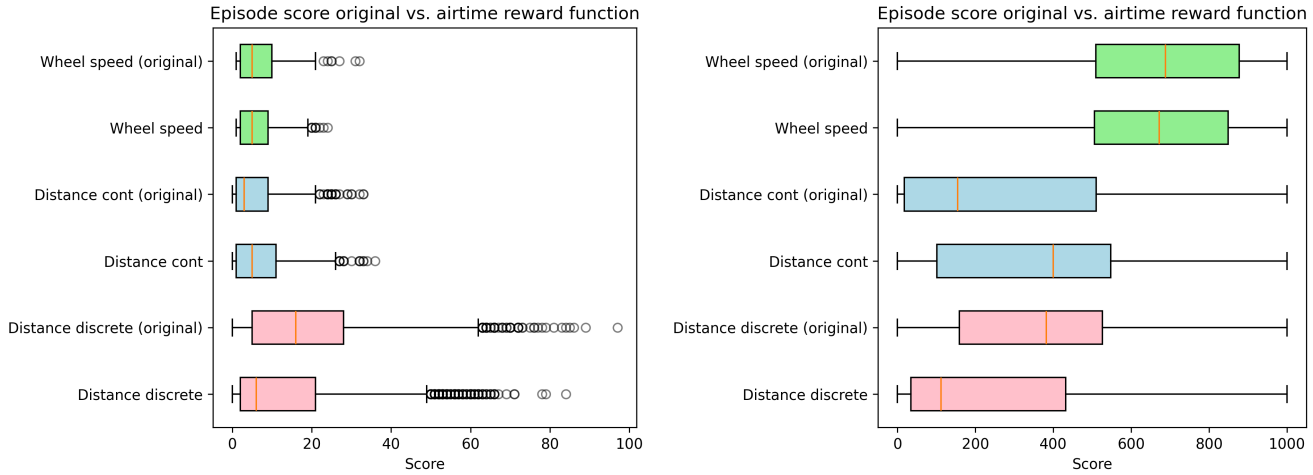


Figure 20: Box plot illustrating the episode scores and total airtime during the training of different agents utilising the airtime reward function compared to the original reward function. The orange line represents the median, and 'cont' denotes the agent's action space to be continuous

6 Conclusions and Further Research

Our primary research question for this thesis was to discover an RL agent that maximises the score as high as attainable in an HCR environment by designing an optimal reward function and finding the optimal action space. From our experiments, the wheel speed-based reward function agent within a continuous action space environment performed the best in the standard environment with a mean score of 773 (see Table 1). However, the same agent achieved an even higher score in an environment where the difficulty increases until the end. The agent consistently reached the 1000 maximum score after only 200,000 training timesteps, as seen in Figure 18. In conclusion, the continuous action space, in combination with the wheel speed-based reward functions, performed the best by far and outperformed any other combination. Although this agent is the slowest driving agent, it is five times slower than the action-based soft agent in the discrete action space environment. Further research is required to improve reward functions, or environmental variables, such as stricter time limits, could be changed to boost the agent's speed.

Moreover, the airtime reward functions did not increase the airtime of the agent at all. Instead, the airtime count was lower; additional research on the airtime reward functions is required to improve the airtime of the agent. The discrete action space could likewise be improved by researching novel distinct reward functions, as our best-performing reward function in the discrete action space only achieved a mean score of 574. Furthermore, another possibility is to research the same environment with the identical terrain generation algorithm HCR utilises to see whether our current Perlin noise terrain generation performs worse or better. Lastly, another algorithm could be utilised than PPO, such as Q-learning with grid discretization of the observation space for the discrete action space agent. Also, our experiments did not optimise PPO hyperparameters, requiring additional research to achieve better performance. Additionally, an experiment with human benchmarks on the HCR environment is possible. The environment contains a human-playable version, which could be utilised in a human benchmark experiment where the scores of humans are observed; this human benchmark can then be used for comparison against the RL agents. Lastly, experiments with a higher maximum score could be conducted. However, this would cost additional computing power for the terrain generation as most experiments already took a long time.

References

[1] L. Zeng, X. Yin, Y. Li, and Z. Li, "Exploring the landscapes and emerging trends of reinforcement learning from 1990 to 2020: A bibliometric analysis," in *Advances in Swarm Intelligence* (Y. Tan and Y. Shi, eds.), (Cham), pp. 365–377, Springer International Publishing, 2021.

[2] L. Zhao, L. Zhang, Z. Wu, Y. Chen, H. Dai, X. Yu, Z. Liu, T. Zhang, X. Hu, X. Jiang, X. Li, D. Zhu, D. Shen, and T. Liu, "When brain-inspired ai meets agi," *Meta-Radiology*, vol. 1, no. 1, p. 100005, 2023.

[3] A. G. Barto and S. Mahadevan *Discrete Event Dynamic Systems*, vol. 13, no. 1/2, p. 41–77, 2003.

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[5] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023.

[6] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, "Reward machines: Exploiting reward function structure in reinforcement learning," *J. Artif. Intell. Res.*, vol. 73, pp. 173–208, 2020.

[7] T. Vu and L. Tran, "Flapai bird: Training an agent to play flappy bird using reinforcement learning techniques," *CoRR*, vol. abs/2003.09579, 2020.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[9] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," 2019.

[10] M. S. Holubar and M. A. Wiering, "Continuous-action reinforcement learning for playing racing games: Comparing spg to ppo," 2020.

[11] L. Janssens, "Crossy road ai: How will the chicken cross the road?," *Thesis Bachelor Data Science and Artificial Intelligence, LIACS, Leiden University*, 2023.

[12] A. W. Moore, "Efficient memory-based learning for robot control," tech. rep., University of Cambridge, 1990.

[13] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems* (D. Touretzky, M. Mozer, and M. Hasselmo, eds.), vol. 8, MIT Press, 1995.

[14] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 1054–1054, 1998.

[15] A. Bădică, C. Bădică, M. Ivanović, and D. Logofătu, "Experiments with solving mountain car problem using state discretization and q-learning," in *Intelligent Information and Database Systems* (N. T. Nguyen, T. K. Tran, U. Tukayev, T.-P. Hong, B. Trawiński, and E. Szczerbicki, eds.), (Cham), pp. 142–155, Springer International Publishing, 2022.

[16] S. Teja Chavali, C. Tej Kandavalli, T. M. Sugash, and J. Amudha, "Modelling a reinforcement learning agent for mountain car problem using q – learning with tabular discretization," in *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*, pp. 1–5, 2022.

[17] W. B. Knox, A. B. Setapen, and P. Stone, "Reinforcement learning with human feedback in mountain car," in *2011 AAAI Spring Symposium Series*, 2011.

[18] Y. Gu, Y. Cheng, C. L. P. Chen, and X. Wang, "Proximal policy optimization with policy feedback," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 7, pp. 4600–4610, 2022.

[19] "Hill climb racing - wikipedia." https://en.wikipedia.org/wiki/Hill_Climb_Racing. [Accessed 24-05-2024].

[20] "Hill climb racing in the app store." https://apps.apple.com/nl/app/hill-climb-racing/id564540143. [Accessed 24-05-2024].

[21] E. G, "Github - code-bullet/hill-climb-racing-ai: Using neat to train an ai to play the classic hill climb racing game — github.com." https://github.com/Code-Bullet/Hill-Climb-Racing-AI. [Accessed 25-05-2024].

[22] "A.I. LEARNS to Play Hill Climb Racing — youtube.com." https://www.youtube.com/watch?v=SO7FFteErWs&ab_channel=CodeBullet. [Accessed 25-05-2024].

[23] L. Weng, "A (long) peek into reinforcement learning," *lilianweng.github.io*, 2018.

[24] Q. Huang, "Model-based or model-free, a review of approaches in reinforcement learning," in *2020 International Conference on Computing and Data Science (CDS)*, pp. 219–221, 2020.

[25] "GitHub - Hill Climb Racing Gymnasium Environment for thesis." https://github.com/alexzh3/hillclimbracing/tree/thesis.

[26] K. Perlin, "An image synthesizer," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, (New York, NY, USA), p. 287–296, Association for Computing Machinery, 1985.

[27] "GitHub - caseman/noise: Perlin noise library for Python — github.com." https://github.com/caseman/noise. [Accessed 03-06-2024].

[28] I. Y. Gkioulekas, "Physics-based Renderin, Lecture 5, slide 38, CARNEGIE MELLON UNIVERSITY." http://graphics.cs.cmu.edu/courses/15-468/lectures/lecture_05.pdf. [Accessed 04-06-2024].

[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[30] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, pp. 229–256, 1992.

[31] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[32] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

Appendices

A Terrain generation code

```
class Ground:
    def __init__(self, world: b2World = None):
        self.ground_vectors = []
        self.distance = hill_racing.GROUND_DISTANCE  # Max distance of the world in pixels
        self.smoothness = 15
        self.grass_thickness = 5
        self.steepness_Level = 0

def randomize_ground(self, seed: Optional[int] = None):
    if seed is not None:
        random.seed(seed)
    ground_seed = random.uniform(0, 100000)  # Generates a random seed that will define the terrain
    # Minimum height of ground
    min_height = 30
    # Initialize a variable to store the additional height for flat ground
    flat_length = 500
    # Initialize a variable to store the additional height
    height_addition = 0

    # Iterate over a range from 0 to self.distance with a step size of self.smoothness
    for i in range(0, self.distance, self.smoothness):
        # Calculate the steepness level by remapping the current distance to a height
        self.steepness_Level = np.interp(i, [0, self.distance], [130, 250])
        # Calculate the noised_y value using Perlin noise with the starting point and adjusted i value
        noised_y = abs(
            noise.pnoise1(ground_seed + (i - flat_length) / (700 - self.steepness_Level), octaves=4))
```

```
        # Determine the maximum and minimum heights for the ground vector based on the steepness level.
        max_height = hill_racing.DIFFICULTY + np.interp(self.steepness_Level, [0, 200], [0, 320])
        # If value is less than the flat section length, recalculate noised_y and height_addition
        if i < flat_length:
            noised_y = abs(noise.pnoise1(ground_seed, octaves=4))
            height_addition = (flat_length - i) / 25
        # Create the ground vector with x-value i and adjusted y-value based on noised_y
        self.ground_vectors.append(
            b2Vec2(i, hill_racing.SCREEN_HEIGHT -
                   np.interp(noised_y, [0, 1], [min_height, max_height]) + height_addition))
```

Listing 1: Relevant code that generates the ground in the HCR environment

B Environment spaces

B.1 Observation spaces learning curve

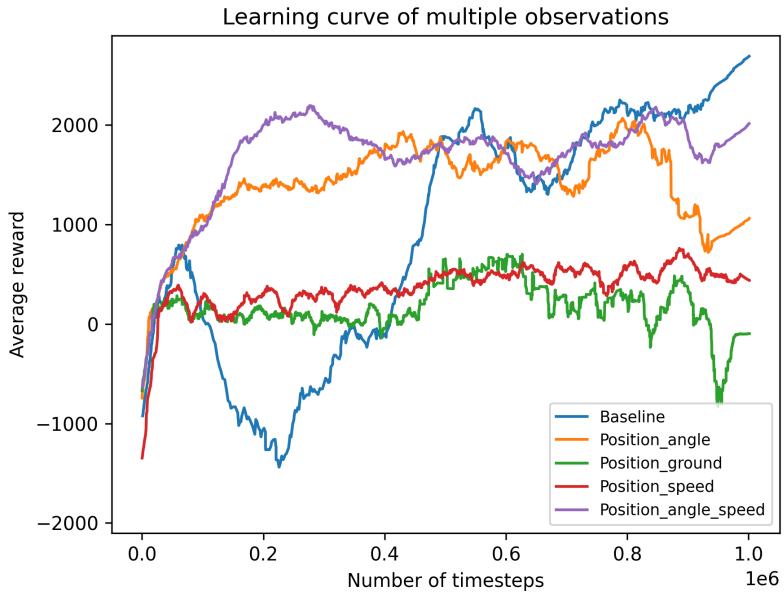


Figure 21: A graph showing the learning curve of different types of observation spaces using PPO. The baseline is defined as all observations (position, angle, ground and speed).

B.2 Observation spaces code

```
self.observation_space = spaces.Dict(
    {
        # x coordinate from 0 to 1000 and y from 0 to 700.
        "chassis_position": spaces.Box(low=np.array([0, 0]), high=np.array([1000, 700]), shape=(2,),
                                       dtype=np.float32),
        # Angle in degrees, can be -36000 to 36000.
        "chassis_angle": spaces.Box(low=0, high=360, shape=(1,), dtype=np.float32),
        # Wheels speed, back and front wheel have same speed limits, add 0.1 to avoid precision errors
        "wheels_speed": spaces.Box(low=-13 * math.pi + 0.1, high=13 * math.pi + 0.1, shape=(2,),
                                   dtype=np.float32),
        # When one of the wheels is makes contact with the ground, 0 means no contact and 1 means contact
        "on_ground": spaces.MultiBinary(n=2)
    }
)
```

Listing 2: Code which shows the observation space's definition in Python

B.3 Action space code

```
# Define action spaces
match self.action_space_type:  # For experiments
    case "discrete_3":
        self.action_space = spaces.Discrete(n=3,
                                             start=0)  # 3 do-able actions: gas, reverse, 3rd action is idling
    case "continuous":  # Continuous motor wheel speeds
        self.action_space = gym.spaces.Box(low=-13, high=13, shape=(1,), dtype=np.float32)

# Execute the action
def _execute_action(self, action):
    match self.action_space_type:  # Check which action space type we have
        case "discrete_3":
            match action:
                case 0:  # Idle
                    self.agent.car.motor_off()
                case 1:  # Gas
                    self.agent.car.motor_on(forward=True)
                case 2:  # Reverse
                    self.agent.car.motor_on(forward=False)
        case "continuous":  # Continuous motor wheel speeds
            self.agent.car.set_motor_wheel_speed(action[0])
```

Listing 3: Code which shows the action space's definition and execution function in Python

C Additional results

C.1 Continuous action space

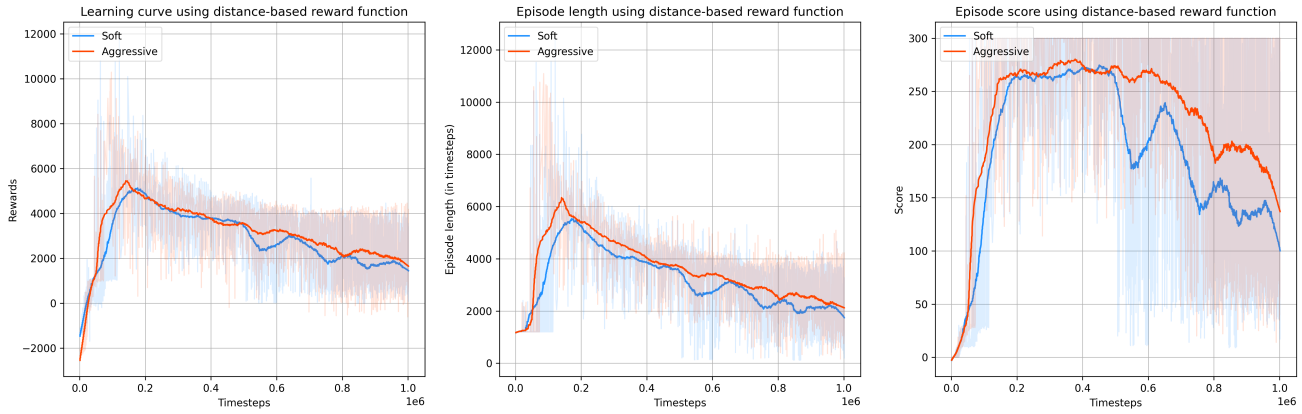


Figure 22: Graphs illustrating the performance of the continuous distance-based reward function experiment in an environment with a maximum achievable score of 300 during the training phase.

Figure 23: Graphs illustrating the performance of the continuous wheel speed-based reward function experiment in an environment with a maximum achievable score of 300 during the training phase.

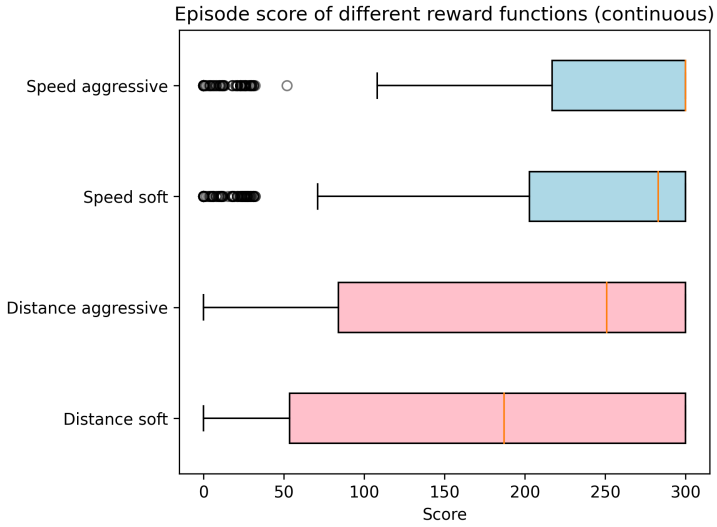


Figure 24: Box plot illustrating the episode scores of a PPO agent in the training phase utilising the distance and wheel speed reward functions in the HCR environment. The experiment utilises the continuous action space and a 300 maximum score. Speed represents the wheel speed reward function, and the orange line in the bar represents the median.

C.2 Difficulty increase versus original training graph

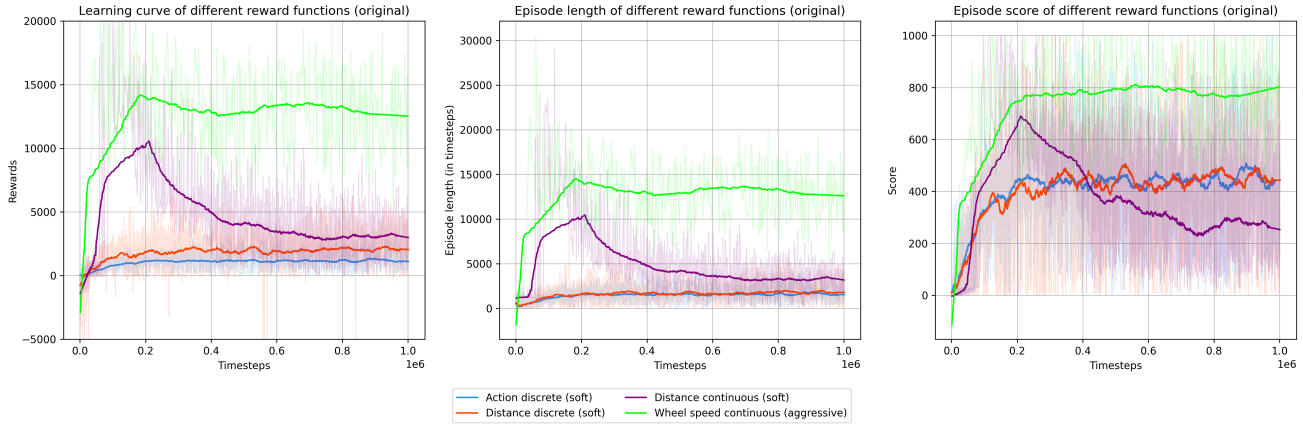


Figure 25: Graph representing the reward learning curve, episode length and score of different agents being trained in the **original** environment.

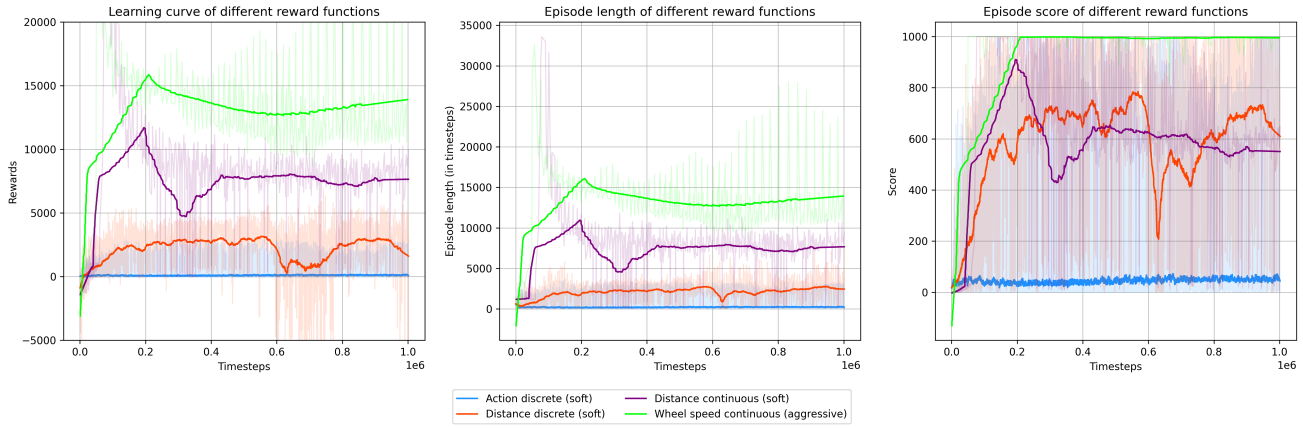


Figure 26: Graph representing the reward learning curve, episode length and score of different agents being trained in an environment where the terrain difficulty increases until the end.

C.3 Airtime versus original training graph

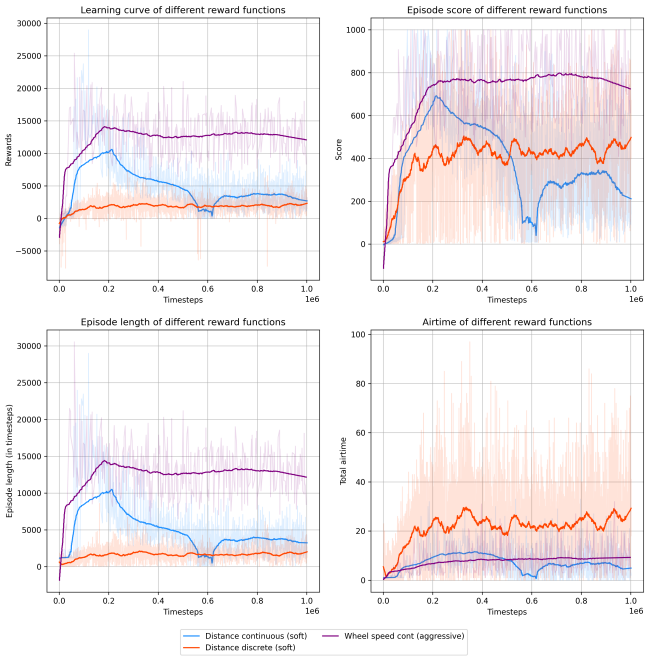


Figure 27: Graph representing the reward learning curve, episode length, score and total airtime of different agents being trained in the **original** environment.

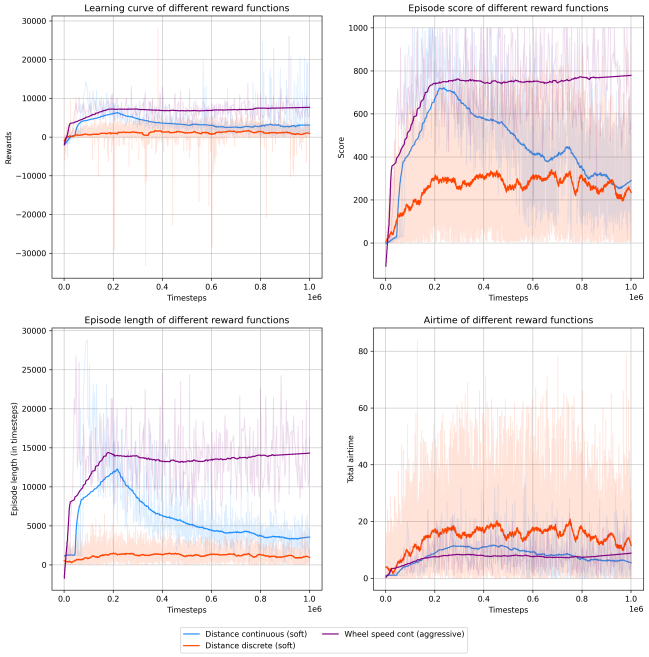


Figure 28: Graph representing the reward learning curve, episode length, score and total airtime of different agents being trained in the environment utilising the airtime reward functions.