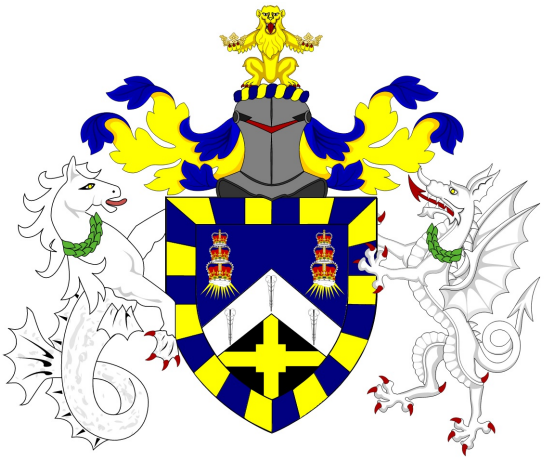
Data Analytics MSc Dissertation MTHM038, 2024/25

Reward Shaping in Reinforcement Learning

Konrad Damian Handke, ID 210451506

Supervisor: Dr. Lennart Dabelow



A thesis presented for the degree of
Master of Science in Data Analytics

School of Mathematical Sciences
Queen Mary University of London

Declaration of original work

This declaration is made on August 31, 2025.

Student's Declaration: I Konrad Damian Handke hereby declare that the work in this thesis is my original work. I have not copied from any other students' work, work of mine submitted elsewhere, or from any other sources except where due reference or acknowledgement is made explicitly in the text. Furthermore, no part of this dissertation has been written for me by another person, by generative artificial intelligence (AI), or by AI-assisted technologies.

Referenced text has been flagged by:

1. Using italic fonts, **and**
2. using quotation marks "...", **and**
3. explicitly mentioning the source in the text.

Abstract

This project examines how different reward structures affect an agent's ability to learn and generalise behaviour. The algorithms: Deep Q-Networks, Double Deep Q-Networks, Duelling Networks, and Prioritised Experience Replay are compared against each other and then optimised for the following two environments. First, the Mountain Car environment is tested alongside a version with a modified reward structure that encourages quicker learning and generalisation. The new reward structure led to improved agent behaviour, but generalisation failed in both versions. After, an environment for temperature regulation is designed with two versions that have distinct reward structures. One version is guided and explains the optimal policy for the environment, while the other version is less explicit. Both versions resulted in the agents correctly completing the task. The less guided version took longer to converge, but performed better in more unseen environments.

Contents

Chapter 1

Introduction

Reinforcement learning (RL) is a branch of machine learning that steers away from traditional supervised and unsupervised learning methods. In supervised learning, labelled data is used to find patterns between inputs and specific outputs. In unsupervised learning, unlabelled data is used to find patterns within the data itself. In RL, an agent uses trial and error to learn the best actions in an environment based on rewards that act as feedback. The main goal of this agent is to find a policy that obtains the highest cumulative reward in the environment. This task is modelled as a Markov decision process (MDP), Chapter 3 of [11], which describes these interactions as a set of actions, states, rewards, and transition probabilities.

1.1 Motivation

In recent years, RL has garnered attention because of its positive applications in fields such as robotics and simulation. In particular, Spot, the agile mobile robot from Boston Dynamics, was trained on various environments to vastly improve navigation through complex terrain [1]. Google DeepMind produced AlphaZero, which was one of the first RL agents capable of competing with state-of-the-art game engines for chess, shogi, and Go [10]. These examples

demonstrate the effectiveness of RL in tasks where it is possible to create an agent-environment interaction. However, RL is still underused in many fields that could see real benefits for reasons including high computational demand, complexity and regulation. Furthermore, various training methods require careful tuning to yield intended or effective behaviour, and each algorithm does not produce the same results across different environments.

This project was motivated by a goal to develop a better understanding of modern machine learning, specifically RL. Due to the complexity of the topic, there are many concepts that can be studied, ranging from different learning algorithms to environmental design. Researching this topic will provide a sufficient challenge while giving insight into how RL can be used in real-life control problems.

1.2 Aims

This project aims to apply a range of RL training methods across different environments with varying reward structures. This specifically includes:

- The implementation and comparison of Deep Q-Networks and commonly used variants.
- Training an agent on the Mountain Car environment, including an iteration with a reconfigured reward structure.
- Designing an environment based on a temperature regulation task and training agents to solve it.
- Analysing different environments based on reward structures by evaluating training and generalisation ability.

Chapter 2

Reinforcement Learning Theory

In this chapter, important RL theory is presented, including the MDP, the Bellman equation, and recent innovations such as Deep Q-Networks (DQN) and variants.

2.1 Markov Decision Process

As mentioned in chapter 1, RL tasks are modelled as an MDP, Chapter 3 of [11]. This model breaks down each environment interaction as a set of discrete time steps t. In each time step, the model features:

- The current state of the environment, denoted as $S_t \in \mathcal{S}$ where $\mathcal{S}$ is the state space. The current state is given to the agent so an action A_t can be produced. The state can be written as a combination of multiple observations, such as (height, length).
- The action the agent took, denoted as $A_t \in \mathcal{A}$ where $\mathcal{A}$ is the action space. The action takes place in the current state to produce the next state S_{t+1} and a reward R_{t+1}.
- The reward, which is a real value denoted by $R_{t+1} \in \mathcal{R}$. This is the feedback received from an environment after action A_t in state S_t.

- The transition probabilities, $p(s', r|s, a)$, where s', r, s, a are the next state, reward, current state, and action, respectively. This expression represents the probability of moving to the next state s' and receiving a reward r.

Once an environment is modelled as an MDP, the goal of an RL agent is to find a policy, π, that maximises the expected cumulative reward, Chapter 3 of [11]. A policy is the conditional probability of selecting an action in a state, $\pi(a|s)$. In many environments, the agent is given a task that does not end, which leads to the inability to maximise the reward as the expected reward sum is equal to infinity. A solution to this problem is to introduce a discount rate, γ, where $0 \leq \gamma \leq 1$, which decreases the value of future rewards. The return at time t can now be defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{2.1}$$

With this return function, the action-value function can be written as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a], \tag{2.2}$$

which denotes the expected return for an action a and state s while following a policy π. This function applies to any policy, regardless of whether it is optimal or not.

2.2 The Bellman Equation

To find the optimal policy in an environment, the Bellman optimality equation, Chapter 3 of [11], is derived from the action-value function, Equa-

tion 2.2. This equation defines the optimal action-value function, $q*$, as

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')], \end{aligned} \tag{2.3}$$

where a' is the action taken in the next state s'. This equation calculates the maximum expected return from a state s after action a, assuming the agent is optimal during the next action a' and after. In this project, the chosen learning algorithms are a set of Deep Q-learning methods, which approximate the solution to the Bellman optimality equation, Equation 2.3 [5].

2.3 Action Selection

For efficient learning, an agent must be able to explore a range of actions so it can discover the best policy in an environment. The action selection method in this project is the ϵ-greedy strategy, Chapter 2 [11]. Here, actions are either random (exploration) with probability ϵ, or greedy (exploitation) with probability $1 - \epsilon$, where $0 \leq \epsilon \leq 1$. Greedy actions refer to actions that maximise the cumulative reward, defined as $\arg\max_a Q_t(a)$ where Q_t is the prediction by the agent of the optimal policy at time t. Values of ϵ are initialised at 1 and decay over a certain number of steps to a minimum value [5]. The minimum is dependent on the difficulty of finding the optimal policy. This forces the agent to start with exploration and eventually proceed into exploitation, where a policy is refined. This project features linear decay, where ϵ decreases to a minimum by a constant value each step, where the constant is equal to

$$\frac{\epsilon_{start} - \epsilon_{min}}{\text{decay steps}}. \tag{2.4}$$

Once ϵ reaches its minimum value, it remains fixed until the end of training. This leaves the agent with a level of exploration, which means it should not

be able to get stuck with a suboptimal policy.

2.4 Algorithms

2.4.1 Q-learning

A standard implementation in RL is the Q-learning algorithm, Chapter 6 [11]. This is a tabular method designed to work in discrete state and action spaces, written as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)], \quad (2.5)$$

where α is the learning rate. The expression $R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ represents the temporal difference (TD) error. This is the difference between the calculated and predicted action-values. During training, the TD error for each state and action combination approaches zero, leading to the action-values converging. Once the action-values have converged, the action with the highest action-value in a state is optimal. This is a method that works well in environments such as Blackjack or Cliff Walking due to their small state and action spaces. However, this method is limited in scalability due to its tabular implementation. To address this, neural networks were incorporated with Equation 2.3 to create DQN [5].

2.4.2 Deep Q-Networks

A neural network is a model composed of an input layer, one or more hidden layers, and an output layer. Each layer consists of nodes that contain weights and biases, which attempt to map a relationship between the input and output layers. The layers are split by an activation function that introduces non-linearity, which enables more complex relationships to be mapped. The model's parameters are updated through backpropagation, which for DQN

and its variants involves minimising the TD error. In this project, the neural networks and the following RL algorithms are implemented in Python using the PyTorch [8] and NumPy [3] libraries.

DQN uses a neural network as a function approximator with weights θ called a Q-network [6]. The Q-network takes a vector of state observations as an input and returns action-value estimates for each action. Due to this, the Q-network allows the use of continuous or high observation state spaces, which were previously not possible with Q-learning. Furthermore, learnt agent behaviour can be generalised, as the Q-network can approximate action-values for unseen states that are similar to trained states. It is important to mention that DQN is incapable of handling continuous action spaces, since a neural network cannot produce action-values for an infinite number of actions.

As mentioned, DQN aims to reduce the TD error, which is done by minimising the loss function

$$L_i(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i))^2], \tag{2.6}$$

where

$$y_i = \begin{cases} r, \text{if terminated,} \\ r + \gamma \max_{a'} Q(s', a'; \theta^-), \text{else.} \end{cases} \tag{2.7}$$

Here, Q, r, a' is the action-value function calculated by a Q-network, the reward, and the action taken in the next state [6]. Termination is the premature end of an episode (a sequence of steps from an initial state to a terminal state, Chapter 3 [11]) where no next states or actions are required since they are not intended. In Equation 2.7, a target Q-network is defined with parameters θ^- [6]. A target Q-network is an identical copy of the main Q-network, but with fixed parameters that are updated to match the main Q-network every n steps. This is done to improve training stability. The

main Q-network is updated through gradient descent as

$$\theta_{i+1} \leftarrow \theta_i + \alpha \nabla_\theta L(\theta_i) \tag{2.8}$$

where α is the learning rate.

Furthermore, DQN introduces experience replay [6]. During training, the steps an agent takes are stored as experiences e_t into a replay memory $\mathcal{D}$. Experiences are defined as $e_t = (s_t, a_t, r_t, s_{t+1})$ while the replay memory is given by $\mathcal{D} = \{e_1, \ldots, e_N\}$, where N is the size of the replay memory. Minibatches of e_t are then uniformly sampled from $\mathcal{D}$ and the Q-network is updated through Equation 2.8. There are several benefits to experience replay, notably the reduction in update variance and increased learning efficiency [5]. If the Q-network receives data from connected time steps, it focuses on updating for patterns in the episode rather than generalising for the entire environment. Once the episode ends, connected samples from the next episode can lead to high update variance. Using random sampling prevents this, as the network trains on a variety of different states. Further, minibatches allow for several θ updates at the same time, which decreases the total training time.

2.4.3 Double Deep Q-Networks

While DQN introduced several innovations, there are improvements to the algorithm that have shown benefits to learning. A problem with the DQN algorithm is the tendency to overestimate the action-values [13]. In Equation 2.7, the target Q-network is responsible for evaluating the next actions, and then the max operator selects the highest action-value. This is a problem as the overestimated action-values are likely to be selected, leading to the network connecting poor actions with positive results. To prevent this,

Double Deep Q-Networks (DDQN) [13] rewrites Equation 2.7 as

$$y_i = \begin{cases} r, \text{if terminated,} \\ r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta_i); \theta^-), \text{else.} \end{cases} \tag{2.9}$$

The idea is to remove the single network evaluation and selection process. Now the main Q-network is responsible for choosing the action, and then the target Q-network evaluates that action. This method has been shown to reduce the overestimation problem [13], but does not eliminate it since the Q-networks are copies of each other. Moreover, tests on a mixture of Atari games have shown better results than DQN on average [13].

2.4.4 Prioritised Experience Replay

Another possible improvement to the DQN algorithm involves replacing the experience replay with a Prioritised Experience Replay (PER) [9]. This replaces the sampling method and changes how θ is updated. Here, an experience is sampled with probability

$$P(i) = \frac{p_i^\tau}{\sum_k p_k^\tau}, \tag{2.10}$$

where p_i is the priority for each experience. The value of τ dictates the level of priority given to each experience, and $\tau = 0$ corresponds to uniform sampling. There are different ways to compute p_i, but this project only features proportional prioritisation where $p_i = |\delta_i| + \epsilon$. Here, δ_i represents the computed TD error for an experience while ϵ is a small error value that is responsible for small or zero δ. The first experience is assigned a priority of 1, $p_1 = 1$. Experiences after are assigned a priority based on the highest current priority in the replay memory, $p_i = \max(p_1, \ldots, p_{i-1})$. New experiences are given a high priority to increase the probability of immediately sampling them into minibatches. This leads to the experiences receiving their correct

TD error, which reflects their real priority. Experiences with a high TD error should be sampled more often, which should increase learning efficiency as the agent trains on more important states.

As mentioned, PER alters how the main Q-network is updated [9]. Each experience is weighted based on its sampling priority, denoted as

$$w_i = \frac{(N \cdot P(i))^{-\beta}}{\max_i w_i}. \tag{2.11}$$

Here, N refers to the current size of the replay memory while β controls the level of importance for each experience. Values of β range from $[0, 1]$, usually starting around 0.5 due to early learning instability and reaching 1 by the end of training. The gradient descent loop is then changed to

$$\theta_{i+1} \leftarrow \theta_i + \alpha w_i \nabla_\theta L(\theta_i). \tag{2.12}$$

PER has been shown to be more effective than experience replay [9]. In this project, PER is used in combination with DDQN.

2.4.5 Duelling Networks

A fourth method of potentially improving DQN is through changing the architecture of the Q-network [14]. The idea is that actions in some states in certain environments do not affect the cumulative reward, suggesting that they are less important. To address this, it is possible to split up the lower layers of a Q-network into connected state-value and advantage layers, written as

$$Q(s, a; \theta, \lambda, \phi) = V(s; \theta, \phi) + (A(s, a; \theta, \lambda) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \lambda)). \tag{2.13}$$

Here λ, ϕ represent the parameters of the different network layers, while V is the state-value function and A is the advantage function. The state-value

function [14] is defined as

$$V^{\pi}(s) = \mathbb{E}[Q^{\pi}(s, a)], \tag{2.14}$$

where $Q^{\pi}(s, a)$ can be referred to as Equation 2.2. The state-value function represents the worth of being in a particular state. With this, the advantage function [14] can be defined as

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s). \tag{2.15}$$

The advantage function represents the importance of each action, measured by subtracting a state's value from the value of the actions in that state.

The latter part of the Equation 2.13 represents the average advantage over each action, where $\mathcal{A}$ is the dimension of the action space. By reducing the advantage by its mean, the state and advantage values combine to produce unique action-values. This makes it easier for the Q-networks to separate the two functions. Further, the advantage above zero is seen as positive to the state, while an advantage below zero is worse for it. This method outperforms the use of a single connected network in many environments [14]. This project uses the Duelling architecture in combination with DDQN.

Chapter 3

Mountain Car

In this chapter, the Mountain Car environment is discussed, including a description of the environment, training, and testing. In this project, every environment was made in Python using Gymnasium [12] and NumPy [3].

3.1 Environment

The Mountain Car environment [2] is a classic control problem that features a car at the bottom of a valley whose goal is to reach the top in the shortest amount of time.

The environment consists of a continuous state space that observes the velocity of the car, ranging from $[-0.07, 0.07]$, and its position along the x-axis, ranging from $[-1.2, 0.6]$. The goal is located at 0.5, and there is a wall at -1.2, which sets the velocity of the car to 0 upon impact. The action space is a set of three discrete actions $\{0, 1, 2\}$, which represent accelerating left, not accelerating, and accelerating right. This closely connects to the physics of the environment, written as

$$\begin{aligned} v_{t+1} &= v_t + f(a-1) - g\cos(3x_t), \\ x_{t+1} &= x_t + v_{t+1}, \end{aligned} \tag{3.1}$$

where v, f, a, g, x denote the velocity, force, action, gravity, and position, respectively. The force and gravity are constants 0.001 and 0.0025. In each episode, the starting position of the car is a random uniform value in $[-0.6, -0.4]$ while the starting velocity is 0. The car then has 200 steps to make it to or past the goal, in which case the episode is terminated, or the episode ends.

This environment has a sparse reward structure where the agent gets -1 per step irrespective of any actions. This means the worst cumulative reward for the episode is -200, but as the episode can end prematurely, this can be reduced. This will be referred to as Mountain Car version 1 (MCV1). As a comparison, another version of this environment was designed with a different reward structure that should be better for generalisation and training. The new reward gives the agent an extra 0.1 for reaching a new highest absolute velocity and a bonus 0.1 for reaching a new furthest distance towards the goal. This will be referred to as Mountain Car version 2 (MCV2).

3.2 Training Methods

The neural network architecture for this environment consists of linear layers with ReLU as the activation function. The input layer takes the state space observation and connects it to a hidden layer with 256 neurons, split by ReLU. The hidden layer then connects to an output layer that produces an action-value for each action. Each algorithm uses this network, except for Duelling, which separates it into the state-value and advantage layers that are combined using Equation 2.13. Smaller networks that have fewer than 256 neurons in the hidden layers converge, but training takes significantly more time.

All parameter optimisation and algorithm selection is done in the MCV1 environment. The best algorithm is found by training 1500 episodes 5 times. Every 50 training episodes, a set of 100 validation episodes run with no θ

updates and no exploration, $\epsilon = 0$. The validation episodes show true agent performance as only greedy actions are chosen. For each validation set, the average reward is computed, and then averaged across the five runs. The standard deviation is also taken across each run, which results in graphs that display the mean performance with shaded error. Every graph was produced in Python using the Matplotlib library [4]. Each algorithm is trained under the same parameters, which should lead to a fair comparison, but it should be noted that the parameters may slightly favour one algorithm over the others. The base parameters are: $\gamma = 0.99$, learning rate = 0.0005, ϵ minimum = 0.01, decay steps = 20000, memory size = 20000, batch size = 128 and the target network is updated every 1000 steps. PER uses the additional parameters: $\tau = 0.7$ and $\beta = 0.5$.

After finding the appropriate algorithm, two loss functions, the mean squared error (MSE) and Huber Loss (HL), are tested. The parameters for the best algorithm are then optimised to get the best agent behaviour. This is done using the same method as algorithm selection. The same parameters are used as a fixed base, while a single other parameter is tested. For example, the learning rate is varied from 0.001 to 0.0001 while the other parameters remain fixed. Each parameter has up to five variants, excluding the discount rate, which is not changed as decreasing it leads to consistently worse results. It is important to state that since the parameters are tested individually, combining them may not be optimal. However, this approach leads to better results than using the base model.

For final training, the procedure remains roughly the same. The best algorithm is combined with its best parameters and trained on the MCV1 and MCV2 environments. Since the model was optimised on MCV1, training the agent on MCV2 may not be optimal, but as the versions are very similar, this should not pose a significant problem. To pick the model for testing, the learning curve that is closest to the mean learning curve is chosen. This is done by comparing each curve by calculating its MSE and choosing the

smallest value. By picking the model closest to the mean curve, any tests will portray the normal agent behaviour.

3.3 Training Results

During training, each algorithm showed interesting results with differences in learning speed and stability. In Figure 3.1, DQN consistently reaches -140 average total reward by episode 400 but declines afterwards until episode 1000, where the agent starts learning beneficial behaviour again. This indicates instability, which is a known issue with DQN. However, at the end of 1500 episodes, DQN does have the highest reward, although each algorithm is in a similar region. Initially, DDQN learns slowly, improving the least for 700 episodes. However, from this point, learning becomes better as the average total reward is consistently higher than the other algorithms. It has the highest variance, meaning it can achieve an even higher average total reward, but it can mean that performance will be worse than other algorithms. PER displays the fastest initial learning, which is expected as the replay memory is prioritised based on the highest TD error. Once the agent reaches the goal there is a high chance the experience is sampled, and the priority is corrected. This experience's new priority will be larger than other experiences since the episode terminates. There is a higher chance that the experience is sampled again, which results in the agent learning sooner. The learning curve is mostly stable past episode 400, resembling DDQN. It is important to mention that PER takes the longest to train in real time. The Duelling curve is very similar to DDQN, but with less variance. As Duelling is used in conjunction with DDQN, it suggests that separating the network architecture may provide stability during training. Despite being more stable, the agent's policy is worse than DDQN throughout training. Disregarding the initial learning phase, each algorithm is similar and returns the same average total reward. This suggests each algorithm can be used to achieve solid per-

formance in the Mountain Car environment. However, DDQN is chosen as it may be possible to consistently reach above −120 reward if the algorithm is optimised. Most likely, similar performance can be achieved by the other algorithms when optimised correctly.

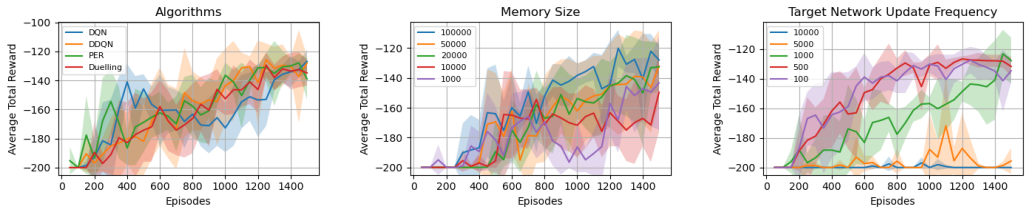


Figure 3.1: The average validation rewards for different algorithms, memory sizes, and network update rates across 1500 training episodes. The points reflect sets of 100 validation episodes that were performed every 50 training episodes. The shaded regions represent the standard deviation of the rewards.

For parameter optimisation, Table 3.1 provides a list of tested and chosen parameters. Graphs corresponding to parameters that are not in Figure 3.1 are in Figure A.2. When testing the two loss functions, MSE is better than Huber Loss by a clear margin. Huber Loss is more deviated and appears to be significantly slower at learning. Out of the five different learning rates, 0.0005 and higher produced nearly identical results, converging to the same point after 1500 episodes. 0.0005 is chosen, but the choice is negligible. From the epsilons, a low minimum of either 0.05 or 0.01 appears best. This suggests that the agent prefers a low rate of exploration to refine a policy, but still wants to explore for a potentially better one. 0.05 is chosen as it deviates to a higher peak, although the choice is again negligible. For the decay steps, the choice is between 100000, 50000 and 20000. These values result in stable training and reach close to the same convergence, making the choice between them insignificant. Figure 3.1 illustrates that there is a single option for memory size, 100000, as it results in the highest reward and most stability. For batch size, as 512 consistently reaches a peak of over −120 total reward, it is selected. The frequency at which the target network is updated will

be every 1000 steps. This is because the learning curve is stable and the reward is consistently highest at the end of training, as shown in Figure 3.1. A frequency of 500 or 100 results in a low variance with a relatively high reward, so it could be chosen. It is worth mentioning that slow rates such as 5000 or 10000 cause the learning process to slow down or stop entirely.

Parameters	**Tested Values**	Final
Loss Function	MSE, Huber Loss	MSE
Learning Rate	0.001, 0.00075, 0.0005, 0.00025, 0.0001	0.0005
Epsilon Minimum	0.1, 0.05, 0.01, 0.001	0.05
Decay Steps	100000, 50000, 20000, 10000, 1000	50000
Memory Size	100000, 50000, 20000, 10000, 1000	100000
Batch Size	512, 256, 128, 64, 32	512
Target Update Rate	10000, 5000, 1000, 500, 100	1000

Table 3.1: Tested parameters for optimisation in the Mountain Car environment.

With the final parameters, the MCV1 agent is trained for 1500 episodes, while the MCV2 agent is trained for 1350. Training on more episodes results in a decline in average agent performance, although some runs may deviate and achieve a higher peak total reward. Due to the difference in reward structures, both agents learn differently, as seen in Figure 3.2. The MCV1 environment does not provide feedback until the agent reaches the goal. As a result, the agent does not learn any behaviour that leads to a different reward for around the first 200 episodes. Once the agent accidentally reaches the goal and the episodes are selected to process, the agent begins to learn. Around episode 1250, the agents' learning plateaus, which indicates they have found a consistent policy to refine. Interestingly, the agent reaches near -100 total reward, but this behaviour is inconsistent and fails to be maintained. By comparison, MCV2 learns the task immediately within the first 50 episodes. However, there is a consistent drop from -150 average reward 50 episodes in to -190 average reward 100 episodes in. This behaviour is interesting and can only be explained by agent exploration. In the beginning, as the agent's

actions are mostly random, the policy changes. MCV2 exhibits a smaller variance across all episodes, suggesting the agents find a consistent path of clear improvement.

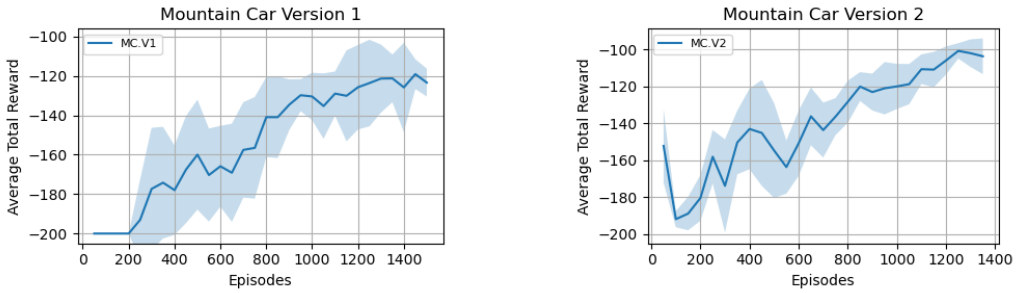


Figure 3.2: The training curves for the MCV1 and MCV2 agents. The points reflect sets of 100 validation episodes that were performed every 50 training episodes. The shaded regions represent the standard deviation of the rewards.

3.4 Generalisation

The goal of testing an agent is to evaluate its behaviour in the normal environment and then to test its ability to generalise to unseen environments. MCV1 may have a slight advantage in the standard Mountain Car environment since MCV2 might want to gather more speed for a higher reward. However, in unseen environments, since MCV2 should learn to gather more speed, it should reach the goal faster. It is important to mention that both agents may fail in unseen environments. Each environment terrain can be found in Figure A.1, and random episodes can be viewed in Figure A.3.

The goal of the Mountain Car environment is to reach the flag in the least number of steps. Since MCV2 was trained using a different reward, comparing rewards directly is not possible. To address this, every test is measured using the MCV1 reward structure since the sparse reward structure represents the total steps used to reach the goal. The method used is similar

to the validation episodes during the training phase, but with an increased number of episodes. The tests are run for a total of 1000 episodes with $\epsilon = 0$. There are two main indicators of performance: the average reward and success rate. The average reward is measured as the average total reward per episode over the 1000 episodes and has up to a ± 2 margin of error. This is equivalent to the number of steps the agent takes. Success is defined as reaching the goal regardless of the total number of time steps. It is calculated as the number of times the agent reached the goal over 1000 episodes. Before evaluating the two agents, it is necessary to discuss the effect that starting positions have on an episode. Depending on the first position, the total reward in an episode can drastically change. This is because the starting position dictates how the agent initially moves the car. For each test, if applicable, the results are separated by ranges in the starting positions.

The first test is conducted on the MCV1 environment. This presents what both agents have learnt and can be used as a benchmark. MCV1 achieved an average reward of -123 with a success rate of $100\% \pm 1\%$ capped at 100%, while MCV2 achieved an average reward of -106 with a success rate of 100%. From this, the average MCV2 agent reaches the goal faster than the average MCV1 agent by around 17 steps. In Figure 3.2, the MCV1 agent can reach -105 reward, but not consistently. This indicates that by adding the extra reward, the MCV2 agent has an easier time learning a better policy. Dividing the starting positions, Figure 3.3 reveals that the difference in the agent's performance depends on the initial state. In the left-most starting positions, $[-0.6, -0.5)$, the MCV1 agent is significantly worse than MCV2. The added rewards make it easier for MCV2 to learn to build less momentum to reach the goal faster. In the middle right starting positions, $[-0.5, -0.45)$, the MCV1 agent is actually better on average than MCV2. In most states, both agents try to go left, then right to the finish in a single swing. At a certain time, the agents must either do nothing or push right to cancel out the momentum left and begin moving to the flag. They attempt to reach a

state from which they know they can reach enough momentum to get to the goal quickly. However, since they do not have enough momentum going left, the agents cancel out the velocity too early. This means that even when the agent pushes to the right, it cannot get to the flag efficiently. The assumption is that both agents attempt this, but as MCV2 is more precise, it attempts this from a lower position, which fails more often. This also explains the large deviation, as the agents have a mixture of success and failure. This can be seen in Figure A.3a, episodes $5, 6, 9$ and 12. In the final range, $[-0.45, -0.4)$, the MCV1 agent outperforms the MCV2 agent but by a few steps. In this bound, MCV1 has a better policy since the deviation is smaller between the scores. As the MCV2 agent gets a reward based on velocity, the agent may predict that gathering more speed leads to a higher reward, which is the reason for the slight average reward difference.

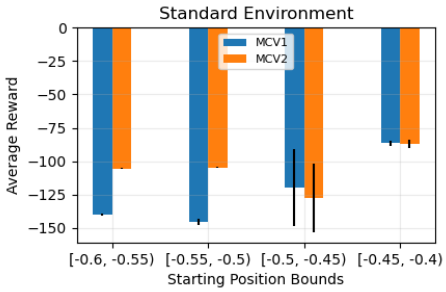


Figure 3.3: The average total reward with error bars for the MCV1 and MCV2 agents across different starting positions in the standard environment.

The second test changes the force in the environment from 0.001 to 0.01, 0.05, and 0.1. This test checks if the agents can complete the task when the outcome of their actions is changed. When increasing the force to 0.01, both agents perform better, which is the expected behaviour as increasing the force should mean the car moves further per step. The MCV1 agent scores -43 average reward while the MCV2 agent scores -34 average reward. Both agents succeed 100% of the time. In the first graph of Figure 3.4, similar behaviour is seen to that in the standard environment. When increasing

the force to 0.05, different behaviour is displayed. The MCV1 agent achieves -41 reward and succeeds 100% of the time, while MCV2 completely fails this task. The agent oscillates between the positions -1.2 and -0.29. The most probable explanation is that the agent is pushing left after hitting the wall, which cancels out any momentum the car has. This would mean the agent gets stuck in a loop where it thinks it cannot make it up the hill. However, there is no reason for the agent to push left when descending the hill. By further increasing the force to 0.1, both agents fail every episode, presumably for the same reason.

Figure 3.4: The average total reward with error bars for the MCV1 and MCV2 agents across different force values.

The third run involves increasing the steepness of the valley by multiplying the latter part of the velocity equation in Equation 3.1 by 1.5. This is equivalent to increasing the gravity to 0.00375. Since the agent may require more time to reach the goal, the total steps are increased to 300. The MCV1 agent scores -139 average reward with a success rate of $95.5\% \pm 0.5\%$ while the MCV2 agent scores -142 average reward with 100% success rate. The strategies that MCV2 used in the standard environment are no longer possible here, meaning it needs to go up the left hill an additional time, which wastes steps. In contrast, MCV1 can use the strategies it developed in the starting positions $[-0.55, -0.45)$. Interestingly, the agent does better than in the standard environment in these starting positions. As it is more difficult to gain velocity, the agent knows that it cannot reach the goal sooner than before, which results in the agent moving back to get velocity sooner. Due

to this, it reaches the goal faster and with more consistency. In the left-most starting positions, MCV1 has a large reward deviation. For an unknown reason, there are occasions when the agent moves around at the bottom of the hill, unable to use its knowledge. This can be observed in episode 2, Figure A.3e.

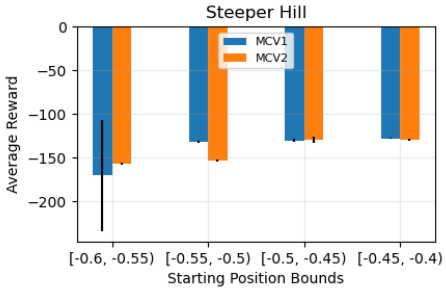


Figure 3.5: Average total reward with error bars for the MCV1 and MCV2 agents across different starting positions in steeper hill environment.

The fourth test is in an environment where the valley is mirrored around the goal. The minimum position is now at 0.4 while the maximum is at 2.2. The starting positions now range from $[1.4, 1.6]$, and the wall is at 2.2. The goal here is to see if the agents can use the same behaviour, but push left instead of right. Unfortunately, both agents fail this task. Initially, the agents gather some momentum, but then begin oscillating between two points. As the agents were trained only in the starting positions $[-0.6, -0.4]$, they are unable to transfer any learnt behaviour into a new environment.

The final test involves increasing the track length by $\frac{2\pi}{3}$ to the left, doubling the valleys. This also includes moving the wall and starting locations to the left. The agent has to climb over the first valley and then continue with normal behaviour from there on. Since this is a longer track, the total steps are increased to 400. Unfortunately, both agents fail this task as well. The agents' behaviour is nearly identical to their behaviour in the fourth test.

Chapter 4

Temperature Regulation

In this chapter, a custom RL environment is designed to regulate temperature in a room. The environment is supposed to replicate a semi-realistic, real-life setting that is inspired by smart home technology. The main purpose of the final environment is to explore and compare different reward structures and how they impact agent learning.

4.1 Design Process

To begin the design process, version 1 (V1) is a simple environment where the goal of the agent is to maintain an inside temperature within a target temperature range. The initial design features three discrete actions: do nothing, open or close the window, and turn a heater on or off. The heater and window were designed as toggles that begin off and then switch on when the agent chooses them as the action. The state space consists of two continuous observations: the inside temperature and target temperature, and two discrete observations: the window and heater status. The inside and target temperatures are randomly generated from a range of values. There is no natural effect that alters the inside temperature. Instead, there is a probability that the temperature either increases or decreases at each step.

The reward structure is divided into a reward that is given when the agent is within a bound of the target temperature, and a penalty if the agent is outside of this bound. The penalty is based on the distance between the inside temperature and the target temperature. It is necessary to establish a range around the target temperature, as it is not possible to maintain a fixed inside temperature. Initial tests on this environment showed positive results as the agent learnt to hold the target temperature.

Version 2 of the environment introduces the outside temperature and increases the number of target temperatures. The goal of this environment is to maintain a target temperature, then move to a new target after h steps. The state space increases in size to four continuous observations: the inside temperature, outside temperature, target temperature, and next target temperature, and three discrete observations: time until the next target temperature, heater status, and cooler status. The next target temperatures are generated by changing the current target temperature by a random uniform value from a range such as $[1, 5]$. The time until the next target temperature is a countdown that informs the agent when the target will change. The action space functions the same, but replaces the window with the cooler. The window needs to be replaced with the cooler as it serves a different purpose. The cooler is an action that reduces the inside temperature by a fixed constant, while the window amplifies the effects of Newton's law of cooling. The window is added in version 3. Since temperature regulation is based on a real-life setting, it is easier to implement features when each step corresponds to an interval of time. For this version and forward, each step is set to 5 minutes, and there will be 288 total steps that cover a 24 hour period. The outside temperature is implemented as a random sinusoidal curve that changes every 5 minutes and covers high and low temperatures across a day and night cycle. With this, it is possible to implement the Law of Cooling. Newton's Law of Cooling states that the rate an object, the inside temperature, changes temperature is proportional to the difference between

the object's temperature and the surrounding environment's temperature, the outside temperature. The Law of Cooling in this environment is given by

$$T_{change} = k(T_{I_t} - T_{O_t}), \tag{4.1}$$

where k is a constant while T_{I_t} and T_{O_t} are the inside temperature and outside temperature at step t. This ensures the environment is more realistic as the inside temperature passively adjusts to the outside temperature at each step. The constant k needs to be tuned carefully as it controls the speed at which an object's temperature adjusts to the surrounding temperature. The constant is set so that the heat loss is realistic in a 5 minute period. This further highlights why the cooler is necessary. If the target temperature is low, but the outside temperature is high, without a fixed decrease in temperature, there would be no way of reaching the target. The value of T_{change} is then used to calculate the next inside temperature in the equation

$$T_{I_{t+1}} = T_{I_t} + T_{change} + H + C, \tag{4.2}$$

where H, C is the power of the heater and cooler. These formulas are used from version 2 until the final version of the environment.

To simulate a more realistic environment, the agent should adjust the inside temperature to the next target temperature before it officially changes. In a real-life scenario, the inside temperature should meet the target temperature when it is requested. This will be referred to as preheating behaviour. To produce this behaviour, there needs to be an incentive for the agent to reach the new target before it switches, as the agent does not naturally pick it up since it is not optimal. This incentive is given in the form of rewards. The versions are split into version 2.0 (V2.0) and version 2.1 (V2.1), which introduce two different strategies to achieve preheating. Since the V1 reward structure works well to get an agent to hold a target temperature, it is used as a base for V2.0 and V2.1. V2.0 implements preheating by rewarding the

agent for being in the next target temperature range when the time until the next target temperature is near the switch. During this time, if the agent is not in range, there is a penalty based on the distance from the inside temperature to the next target temperature. This allows the agent to transition to the next target without receiving multiple penalties. In contrast, V2.0 gives a large reward to the agent when the inside temperature is in the range of the next target temperature when time is 1. This large reward covers the cost of transitioning to the next target temperature. The intention is to create two distinct agents where one agent is explicitly guided with clear rewards, while the other agent tries to find the best policy alone.

Version 3 works on the addition of a window action, which should work as an amplifier to the constant k in Equation 4.1. When the window is open, the rate at which heat transfers should naturally increase since there is airflow between the inside and outside. To facilitate this change, the window is assigned the constant w, which appears before k in Equation 4.1. With the window, the state space is altered to include the window status as a discrete observation. To further improve the environment, the agent should be able to make multiple actions in one step. To do this, the agent is given eight discrete actions, which are the combinations of using the heater, the cooler, and the window. For example, the action $(0, 0, 1)$ represents turning the heater off, the cooler off, and the window open. The reward structures are split into versions 3.0 (V3.0) and 3.1 (V3.1), where V3.0 is a continuation of the V2.0 reward structure and V3.1 is a continuation of V2.1. V3.0 provides extra rewards that explain the states the window should be used in to make use of natural cooling. In contrast, V3.1 gives extra rewards based on an energy system where the heater and the cooler are penalties, and the window is a reward. Since these reward structures are present in the final version of the environment, they are explained in detail in the next section. When testing the environment, there are two recurring problems. The first problem is that the target temperature becomes stuck as its acceptable maximum

or minimum. This should occur, but not at the current frequency. The second problem is that the environment cannot be solved when the target temperatures become stuck while the outside temperature is at the opposite extreme. These issues are addressed in the final version of this environment by redesigning how the outside temperature is modelled. This is described in detail in the next section.

4.2 Environment

In this section, the final version, version 4, of the environment is broken down in detail. The goal of the environment is to maintain an inside temperature at a target temperature and then transition to the next target temperature.

The environment consists of a state space that observes the five continuous variables: the inside temperature $[-11, 30]$, outside temperature $[-6, 25]$, next outside temperature $[-6, 25]$, target temperature $[-11, 30]$, next target temperature $[-11, 30]$, and the four discrete variables: time until next target temperature $\{1, 2, \ldots, 288\}$, heater status $\{0, 1\}$, cooler status $\{0, 1\}$ and window status $\{0, 1\}$. The variable next outside temperature is added to possibly allow the agent to have a better idea of when to begin adjusting the inside temperature to the next target temperature. The action space consists of the same 8 discrete actions seen in version 3.

To fix the issues with the outside temperature, there is no longer a randomly generated sinusoidal curve, but rather four sinusoidal curves that roughly simulate realistic temperatures for each season. To find the temperatures for each season, an analysis was done using England data in "Data tables of UK and regional monthly series" found on the Met Office [7]. This returned four means and standard deviations corresponding to each season. The standard deviations were too small to produce a range of states for the agents to learn, so they were manually adjusted. Each seasons mean was spread apart enough to create four distinct weather patterns. To alleviate the

problem of the agent being unable to reach certain target temperatures, each season is tuned so that the available minimum and maximum temperatures are within a range of the outside temperature curve. Further, the seasons have a fluctuation value, which represents the distance between the highest or lowest value and the mean, and a noise range that shifts the position of the curve. These values are used in the equation

$$(\text{fluctuation} \cdot \sin(x) + \text{mean}) + \text{noise}. \tag{4.3}$$

Here, x is a sequence of 600 values starting from $[3.5, (3.5+4\pi)]$, that displays slightly over a 48 hour period. Only a 24 period or 288 values are seen each episode. The seasonal parameters are in Table 4.1. The season is picked randomly at the start of every new episode.

Season	**Mean**	**Fluctuation**	**Noise**	**Temp Min**	**Temp Max**
Winter	4	5	$[-5, 0]$	-11	14
Spring	8	10	$[-3, 3]$	-4	20
Summer	15	5	$[0, 5]$	5	30
Autumn	10	10	$[-3, 3]$	-2	22

Table 4.1: Seasonal temperature parameters.

Another issue from version 3 is that the target temperatures were consistently reaching and staying at the minimum or maximum temperatures. Instead of keeping the changes truly random, the new version determines the target through a directional bias. If the target increases, the probability of the next target decreasing increases by 10%. The starting probability at each reset is 50%. By doing this, the target temperatures are less likely to become stuck, which should be better for agent learning. Additionally, the next target temperatures are always picked relative to the current target from a range $\pm[4, 6]$. The target temperatures switch every 24 steps or two hours, which is enough time to maintain the temperature while also moving to the next target in time.

The core of how the environment functions is through Equation 4.1 and Equation 4.2. The value of k is very important in the dynamics of the environment. If k is too large, the outside temperature has too much influence over the inside temperature, resulting in many unsolvable states. If k is too small, the influence is negligible, so the agent can ignore the outside temperature completely. In this version, $k = -0.0175$, which ensures the influence of the outside temperature is balanced across every difference of the inside and outside temperatures. As previously stated, the window is an amplifier given by w. Here, $w = 3$, which results in the window creating a clear difference when open, while not overpowering the heater or the cooler. When opening the window, the value of k is treated as -0.0525, corresponding to $w \cdot k$. In Equation 4.2, H, C is equal to $0.5, -0.5$ respectively. If the heater is on for 12 steps, the temperature increases by 6, assuming no influence from the outside temperature.

Version 4 has three distinct reward structures split into versions 4.0 (V4.0), 4.1 (V4.1), and 4.2 (V4.2). V4.0 has a very guided reward structure that is divided into two parts. The first part rewards the agent 0.5 for staying within ± 0.5 of the current target temperature while punishing the agent with the absolute difference between the current target and inside temperature. The second part is only applicable when the time to switch target temperature is less than or equal to 8. The agent gets zero reward for being in range of ± 1.5 of the next target temperature, while punishing the agent with the absolute difference between the next target and the inside temperature. Most transitions between the current and next target temperatures last from around 10 to 6 steps, depending on the outside temperature. If the time is less than 8, the agent overshoots the next target, while increasing the time ends with the agent reaching the next target prematurely. There are additional rewards that are given to the agent. When the window is used while the outside temperature is greater than the inside temperature and the inside temperature is less than the current target or next target temperatures, the

agent is rewarded with an extra 0.25. This is also the case for the opposite scenario. However, the agent is punished -1 for using the heater and cooler together and -0.25 for using the heater or cooler with the window. This setup explains how the window should be used and promotes being energy efficient. It also explains that combining the window with another action is less efficient, but not a negative behaviour.

V4.1 has a reward structure that enables the agent to learn more on its own. The agent is given an energy penalty that rewards the agent -0.5 for using the heater or cooler, and 0.5 for using the window. This energy penalty is given to each following reward. To promote preheating behaviour, the agent is given a large reward of 20 for being in ± 1 range of the next target at time 1 before the switch. This is a delayed reward that forces the agent to receive multiple penalties before receiving a single large positive reward. If the agent plans well, the reward should be enough to cover the cost of transitioning between target temperatures. At all other times, the agent is given zero reward for staying in the target temperature range, ± 0.5. If not in this range, the agent receives a negative reward based on the absolute difference between the current target and the inside temperature. This makes it so the optimal policy is to preheat, but the agent must be precise with its actions. V4.2 is used to show the general behaviour of the agent when no incentive is given for preheating. The reward structure is similar to V4.1, but without the large positive reward and energy penalties.

4.3 Training Methods

The neural network model for this environment consists of linear layers with the ReLU activation function. The input layer receives a vector of the state space explained in section 4.2. This is followed by two hidden layers with 128 neurons each, split between ReLU. The output layer produces an action-value for each action. The architecture is the same for each algorithm, apart

from Duelling networks, which splits the architecture after the input layer. Throughout training versions 1 to 4, the complexity of the neural network has naturally increased since the environment has become more complex. Early versions of the environment could use a single hidden layer, but from versions 3 upwards, as the action space became much larger, two hidden layers had to be used. Testing also showed that fewer than 128 neurons resulted in worse behaviour.

The training setup for this environment is similar to the Mountain Car training seen in section 3.2. The base parameters are changed to: $\gamma = 0.99$, learning rate = 0.0001, ϵ minimum = 0.01, decay steps = 57600, memory size = 28800, batch size = 128 and the target network is updated every 1440 steps. PER uses the additional parameters: $\tau = 0.7$ and $\beta = 0.5$. The training setup consists of 1500 episodes trained only on the V4.1 environment. As a result, the agent's performance in the V4.0 and V4.2 environments may be worse. However, since the dynamics of the environment are the same, the difference should be small. For final training, the method remains the same as Mountain Cars for both the V4.0 and V4.1 agents. The V4.2 agent is not intended to be used in any real testing scenarios and is only used for initial comparisons. Due to this, the agent is only trained a single time under the optimised parameters for V4.1. Training only once does mean that the agent's behaviour could be worse than average, but it is enough to highlight the differences between learning with and without the preheating incentives.

4.4 Training Results

As shown in Figure 4.1, the learning curves for DQN, DDQN, and Duelling are not very different. Learning is mostly stable, with Duelling having the smallest variance and DDQN having the highest, which is similar to the results in Figure 3.1. From these algorithms, steady learning is shown until episode 700, where it starts to plateau. This is interesting, as it may be

expected that three algorithms produce different results. At the very end, DDQN consistently achieves the highest reward near −100, which is roughly 20 better than DQN and Duelling, so it is the selected algorithm. The overestimation bias from DQN does not appear, as the results are similar to DDQN, which solves this issue. The Duelling networks only reduce learning variance, which means the states are of equal importance. There is a clear outlier as PER is significantly worse than the other algorithms. The best average reward is around −150, which was gained from a deviation. As the reward is delayed, the agent receives a series of different negative rewards and only then, if it reaches the bound, a positive reward. When the transition steps are sampled, the difference between the actual and predicted action-values is likely high as the agent cannot figure out whether to maintain or adjust the temperature. Therefore, the priority of the experience remains high, which creates a loop where the agent samples the experience but receives no useful feedback.

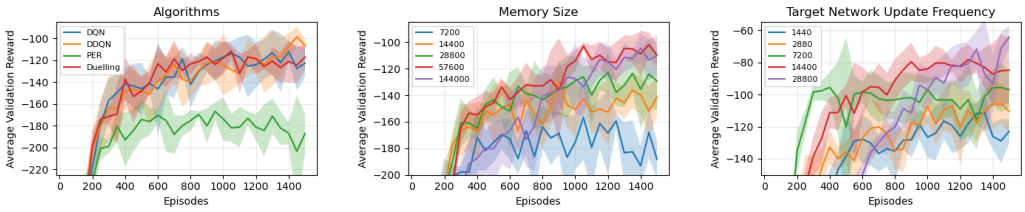


Figure 4.1: The average validation rewards for different algorithms, memory sizes, and network update rates across 1500 training episodes. The points reflect sets of 100 validation episodes that were performed every 50 training episodes. The shaded regions represent the standard deviation of the rewards.

Table 4.2 provides a list of final and tested parameters. Graphs corresponding to parameters that are not located in Figure 4.1 are in Figure B.2. Similar to previous findings, MSE is significantly better than Huber Loss. Even with the added reward for preheating, the agent trained with Huber Loss does not engage with that behaviour. Instead, it chooses to move to the next target temperature when it officially switches. The learning rate curves

suggest that a lower learning rate is better for the agent. Both 0.0001 and 0.00025 are comparable throughout the episodes with similar reward peaks and stability. The higher learning rate is chosen as it may provide faster convergence when training with more optimised parameters, but the choice is likely to yield similar results. A lower epsilon minimum concludes in better training results. Since there are two strategies in this environment, the agent picks up the preheating policy and then wants to refine it rather than continue exploring, as there is nothing else to explore. Examining the decay step curves results in either the choice of 28800 or 57600. As the agent prefers refining the strategy, shown with the low epsilon, getting to the epsilon minimum faster may result in quicker learning. In Figure 4.1, the memory sizes suggest that 57600 is the best option. It displays the highest reward peaks with a low deviation. It is possible to take 144000 since at the end of 1500 episodes it converges to the same reward 57600 does. The different batch sizes mostly result in similar behaviour. A batch size of 256 is chosen since it converges at episode 1000, while the other batch sizes take 1500 episodes. A 64 batch size could be chosen as the total training time is lower than with higher batch sizes. The rate at which the target network updates is the most important parameter, as shown in Figure 4.1. A value of 28800 can be selected as it reaches a consistent peak of -65 reward. However, 14400 is chosen since it is the most stable with a possibility of reaching the same peak reward. Unlike in the Mountain Car environment, a faster rate yields worse results.

With the chosen parameters, the V4.0 agents train for 1000 episodes while the V4.1 agents train for 1500 episodes, as beyond this point, their behaviour starts to decline. As seen in Figure 4.2, both learning curves are very similar. Due to the guided reward, the V4.0 agent converges earlier than V4.1. It is even possible to shorten the training time to only 650 episodes since the improvement from that point is minimal. The strategy is significantly easier to learn and refine, as shown by the low variance across the curve. There is

Parameters	**Tested Values**	Final
Loss Function	MSE, Huber Loss	MSE
Learning Rate	0.001, 0.00075, 0.0005, 0.00025, 0.0001	0.00025
Epsilon Minimum	0.1, 0.05, 0.01, 0.001	0.001
Decay Steps	144000, 57600, 28800, 14400, 2880	28800
Memory Size	144000, 57600, 28800, 14400, 7200	57600
Batch Size	512, 256, 128, 64, 32	256
Target Update Rate	28800, 14400, 7200, 2880, 1440	14400

Table 4.2: Tested parameters for optimisation in the temperature regulation environment.

an instance of high deviation at episode 400 when the behaviour consistently worsens. This is unusual and cannot be explained. In contrast, since the V4.1 environment is less guided, learning takes a longer amount of time. Similar to V4.0, the agent quickly learns the preheating behaviour and then refines it. As it is harder for the agent to reach the required state to receive the large positive reward, the variance across the curve is higher. Identical to V4.0, there is a point at episode 400 where the agent's behaviour gets significantly worse. Again, this cannot be properly explained, as there should be no reason for this to occur in any episode. One potential explanation for this behaviour is that the agent is switching strategy to exclude preheating. However, there should be no reason for this to consistently occur across two different environments at the same episode.

4.5 Generalisation

Testing is similar to the Mountain Car testing. Each environment is run for 1000 episodes with $\epsilon = 0$. Random episodes can be viewed in Figure B.1 and Figure B.3. The V4.0 and V4.1 agents are tested for their ability to generalise, while V4.2 is only compared in the first test. The expectations for these tests are that the V4.1 agent should adapt better, as less guidance

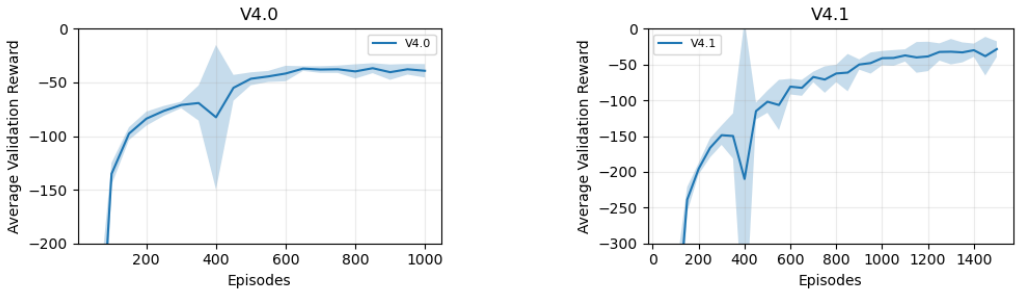


Figure 4.2: The training curves for the V4.0 and V4.1 agents. The points reflect sets of 100 validation episodes that were performed every 50 training episodes. The shaded regions represent the standard deviation of the rewards.

in the reward structure means it must learn the behaviour on its own for each different state. As the V4.0 agent receives a guided reward, it should be more rigid with its actions, resulting in less adaptability. However, V4.0 may perform better in measures such as energy efficiency. Since the V4.0 agent is given specific rewards that explain beneficial window use, the agent should make decisions that reduce heater and cooler use. In V4.1, the agent is given an energy penalty as part of the reward system, but as the window is given a positive reward, the agent may learn to use it in conjunction with the heater or cooler to mitigate the penalty.

The goal of this environment is to maintain the inside temperature within the target temperature bounds and then move to the next target temperature before it officially switches. To measure the goal, the agents are compared using a series of performance metrics. These metrics do not include average rewards, as comparing the results of different reward structures leads to no conclusion. Furthermore, comparing agents using the same reward structure, for example, only on V4.0, can lead to understanding which agent is better, but it could mean the agent is exploiting the reward structure. The first metric measures the total time the inside temperature is within the bounds of the target temperature. This will be referred to as the control score and is calculated as an average per episode. The evaluation of this metric is

dependent on the total number of target temperature switches per episode. If the switch occurs every 24 steps, it means that in an episode, 11 switches occur with 1 extra switch at the end. If the agent preheats to the next target temperature each switch, and this takes an average of 8 steps, a perfect agent should achieve a score of near 65%. The second metric assesses the agents' ability to transition to the next target temperature. This will be referred to as the preheating score. If the inside temperature is within ± 0.5 of the target temperature when the time to switch is at its maximum time, the agent succeeds. In the standard environment, the maximum time is 24. This is calculated as an average per episode. The third metric checks the average time each action component, the heater, the cooler or the window is used. It is measured by taking the amount of time the components are on per episode and then averaging over every episode. The final metric measures the energy efficiency of the agents by assigning the heater and cooler a penalty of 0.5 per use. This is then tallied across all 1000 episodes and measured as an average per step. Not all metrics are discussed in every test, and the first three metrics have up to a $\pm 2\%$ margin of error.

The first test is conducted in the standard environment. The V4.0 agent stayed in the target range 53% of the time with a 68% preheating score. The V4.1 agent secured a control score of 58% with a preheating score of 52%. V4.1 is better at holding the temperature for a longer period of time, which means it has learnt how to adjust the transition time depending on the state. However, it is worse in terms of the preheating score. On average, the V4.0 agent manages to successfully reach the correct range 7 to 8 times per episode, while the V4.1 agent only does it 5 to 6 times. The V4.0 and V4.1 agents consistently attempt this behaviour, but are outside of the ± 0.5 range, meaning they fail the metric. Moreover, the reward structure for V4.1 guides the agent to stay within a ± 1 bound of the next target temperature when the time to switch is 1. This results in decreased performance in the preheating score, as the agent is slightly outside the range. Extending the

preheating range to ±1 results in both agents scoring around 90%, but V4.1 is still worse. As a comparison, the V4.2 agent is in the target range 63% of the time with a 3% preheating score. This highlights the necessity of adding an incentive for the agent to preheat, as this behaviour does not appear naturally. Furthermore, it illustrates that V4.0 and V4.1 should be capable of increasing their control score. In terms of component use, V4.0 used the heater, cooler and window an average of $33\%, 33\%, 48\%$ with an energy per step of 0.33. The V4.1 agent uses the heater and cooler slightly more at 34% each, resulting in more energy per step at 0.34. The window use increases to 65% since there are no explicit instructions on how to use the window optimally.

The second test changes the target switches from every 24 steps to 12 and 144 steps. This evaluates whether the agents can adjust to the time they spend in the target temperature range. The best possible control scores for 12 and 144 steps are roughly 35% and 95% respectively. In the 12 steps test, V4.0 achieves a 15% control score and a 53% preheating score while V4.1 achieves a 26% control score and a 43% preheating score. V4.1 is far better at holding the inside temperature, but struggles in the preheating score. In many episodes, V4.0 fails to meet the target temperatures at all, as shown in Figure B.3a, episode 4, which is not the case for V4.1. The issue arises when the difference between the outside and target temperatures is large. V4.0 does not attempt to meet the targets, while V4.1 does. Another problem is the reward structure that causes the V4.0 agent to transition at 8 steps, which leaves little time to hold the temperature. Further, V4.0 has an energy per step of 0.41 while V4.1 has 0.37. The energy per step is naturally higher than in the standard environment due to the increase in switches, but V4.0 should be more energy efficient than V4.1. The issue is that when V4.0 does not try to reach the target, its actions become random, so there is an increase in energy consumption. When the switches change to every 144 steps, both agents fail to perform well. V4.0 achieves a 26%

control score and a 75% preheating score while V4.1 achieves 24% and 50%. Both control scores are very low as the agents struggle to hold the target temperature for a prolonged period of time. A plausible explanation is that the agents do not use any learnt behaviour until the time reaches 24 or less, which are the values seen in training. When the time is higher, the actions are more random. However, holding the inside temperature at a target is a common behaviour that the agents learn, so it is unclear why this would occur. Another interesting observation is the energy per step, as V4.0 scores 0.57 and V4.1 scores 0.56. This is extremely high and caused by both agents using the combinations of all components turned on at the same time, which is not normal behaviour.

The third test takes place in a very different environment that adjusts the outside temperature and explores values unseen to the agent. The mean, fluctuation, noise, temperature minimum and temperature maximum are $50, 5, [-3, 3], 40, 60$ respectively, while the inside temperature is 50 at the start of every new episode. This test checks if the agents can perform the same behaviour when they receive different observation values. The V4.0 agent achieves a control score of 33% and a preheating score of 53% while V4.1 only scores 20% in both metrics. The V4.0 agent demonstrates good adaptability, being able to behave similarly to how it does in the standard environment. The agent follows the target temperature well, with a minor issue involving being outside of the range when trying to maintain the target temperature. In most cases, the V4.1 agent fails to hold the temperature and acts randomly, as can be seen in Figure B.3c, episodes $8, 9, 10, 12$. When it maintains the temperature, it frequently undershoots or overshoots it during the transition. This is unexpected, as no part of any reward structure should make it easier for the agents to handle unseen temperature values. A possible explanation is that, as the V4.1 agent learns preheating behaviour from scratch, it overfits to the temperature values seen in training, which limits its potential to generalise. In contrast, V4.0 receives a plan to preheat

through its reward structure, meaning it does not need to focus on the temperature values as much, potentially making it easier to generalise. Another interesting aspect is the component usage. V4.0 uses the heater, cooler, and window at rates of $30\%, 26\%, 60\%$, while V4.1 uses rates of $29\%, 37\%, 76\%$. If the agents treat this environment as the standard environment, the rates should roughly be the same. However, this is not the case, and the rates are lower than in the standard environment. An explanation for this is that the difference between the maximum and minimum temperature is 20 rather than 24 or 25 as seen in the standard environment. V4.1 also has a large difference between the cooler and heater rates, which can be explained by the overshooting behaviour.

The fourth test is designed to check if the agents can use their learnt behaviour when the target temperature changes are lowered from $[4, 6]$ to $[0, 4]$. This test assesses whether the agents can maintain the inside temperature for longer and reduce the time it takes to transition to the next target temperature. Taking 4 steps on average to transition, it is possible to achieve a control score of around 83%. The V4.0 agent managed a 61% control score and a 72% preheating score while V4.1 achieved a control score of 72% and a preheating score of 67%. The difference between the control scores is due to the different reward structures. V4.0 guides the agent to always transition 8 steps before the target changes, while V4.1 allows complete agent flexibility. This flexibility allows the agent to hold the target temperature for longer than V4.0. Although V4.0 is outperformed by V4.1, it still adapts well. V4.1 scores 0.26 energy per step while V4.0 uses 0.27 energy per step. Both agents are far more efficient than in the standard environment, which should occur if they generalise well.

The last test adjusts the target temperature changes from $[4, 6]$ to $[0, 2]$ and the switches to every 8 steps. This is a combination of the second and fourth tests that evaluates the agent's ability in a more dynamic environment. The V4.1 agent achieves a control score of 59% and a preheating score of

66% while V4.0 achieves a control score of 32% and a preheating score of 51%. Here, the V4.1 agent is significantly better, beating out V4.0 in both metrics. As mentioned previously, the flexibility of V4.1 allows it to better adapt when the temperature changes and the switching intervals decrease. Further, the preheating rewards for V4.0 cause the agent to continuously chase the next target temperature during times when it should hold the current target. Additionally, V4.1 uses the heater, cooler and window at rates of $22\%, 18\%, 42\%$ whereas V4.0 uses them at rates of $20\%, 29\%, 41\%$. This results in the V4.1 agent being more energy efficient at only 0.21 per step compared with V4.0 at 0.25 per step. Overall, V4.1 is better at adapting to more dynamic environments.

Chapter 5

Conclusion

This project aimed to explore how different reward structures impact an agent's ability to learn and generalise behaviour.

In the Mountain Car environment, the modified reward structure showed improvements over the standard reward in the normal environment. The agents were trained for the same amount of time, but MCV2 found a better policy than MCV1. However, improvements in behaviour come with potential downsides. To get the best results, MCV2 overfits to the standard environment, which is evident in the steeper hill test, where it performs slightly worse than MCV1. This can hurt the agent's ability to generalise, but it can be useful in tasks with a fixed setting. The last two tests in section 3.4 highlight a severe limitation of both agents as they are unable to generalise their behaviour. Although the final goals in the environments are located in the same position as the standard environment, the unfamiliar starting positions render the agents unable to function. Changing the reward structure can improve agent behaviour, but it is not enough to improve generalisation. The environment needs to be designed so that the agents are exposed to a variety of states during training. The problem with the Mountain Car environment is that it is not designed for generalisation. With the current state space, the agent does not observe the goal location, so it cannot be changed. It should

be possible to extend or mirror the terrain during training, which could result in the success of the last two tests. Overall, the Mountain Car experiments showed that reward shaping can result in better agent performance, but not necessarily generalisation.

In the temperature regulation environment, two different reward structures were compared for their ability to learn and generalise. Both reward structures showed different results, with V4.0 taking less time to train and achieving a higher preheating score in the standard environment, while V4.1 achieved a higher control score. Unlike the Mountain Car environment, this environment was designed with generalisation in mind. There were far more states that the agents saw during training, which encouraged the agents to develop a flexible policy. This was shown in the third test, where the V4.0 agent translated its behaviour over and succeeded. In the tests after, the less guided reward in V4.1 showed better generalisation than V4.0. The agent could adjust when to transition over to the next target temperature, which was not possible with V4.0. However, there are issues with the reward structures that decrease testing scores. In V4.1, the agent is permitted to be within a range of ± 1 of the next target temperature before the switch. Reducing this range should improve the agent's precision, but it will make the behaviour harder to learn. A significant issue with V4.0 is that it reaches the next target temperature prematurely. A solution to this problem is to swap the preheating reward with the reward in V4.1. V4.1 is simpler and with more adjustments should achieve far better scores than seen in the tests. One unexpected problem occurred when the switch times were increased. In theory, the agent's behaviour should not alter when the time values increase, as they should not be relevant until the switch is near. In practice, this is one of the most important observations as it tells the agents how to act at a given moment. For example, if the time is in single digits, the agents begin transitioning to the next target temperature. For unseen times, the agents do not know how to respond correctly. To fix this issue, the agents should

be exposed to a variety of switch times during training.

There are also broader issues related to RL that arise in this project. RL takes a significant amount of computational power and time to train and test different environments. Increasing the complexity of the environment results in the need to use a neural network with more layers and neurons. This, in combination with a more difficult environment, means the agent takes more time to converge. For example, a natural progression of the temperature regulation environment is to either add another room or another variable to control, such as humidity. Both modifications increase the state and action space, which increases the difficulty of the task and requires a larger neural network. Furthermore, another issue revolves around generalisation. As shown in the temperature regulation environment, the agent needs to explore a wide range of values in the observations to generalise behaviour to unseen states. Even then, generalisation is not guaranteed and depends on the rewards. In V4.0, the agent can translate its behaviour into the third test environment, while V4.1 cannot. This is then different in the tests after, as V4.1 generalises to smaller temperature changes more effectively. With this in mind, the most effective applications of RL are tasks with a fixed goal that do not aim to achieve any generalisation. Here, the dynamics of the environment are known, so the focus is to create a reward structure that delivers optimal agent behaviour. This would include game environments such as chess [10].

Finally, there is significantly more that can be explored with reward design. This project aimed at testing large changes between reward structures, but small differences, such as increasing the value of the additional rewards in MCV2, could lead to very different results. For future projects, an analysis of how small reward changes impact an agent's learning and behaviour could provide valuable insight into RL agents and lead to improvements in the current temperature regulation environment design.

Appendix A

Mountain Car Graphs

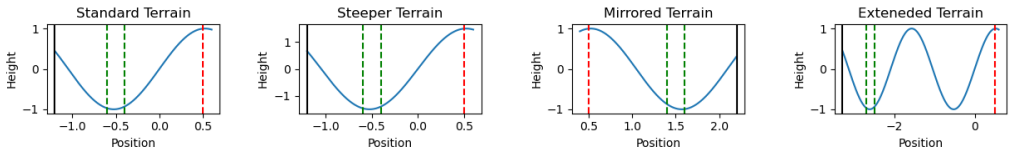


Figure A.1: Terrains of different versions of the Mountain Car environment. The black line is the wall. The green lines are the bounds of the starting positions. The red line is the goal.

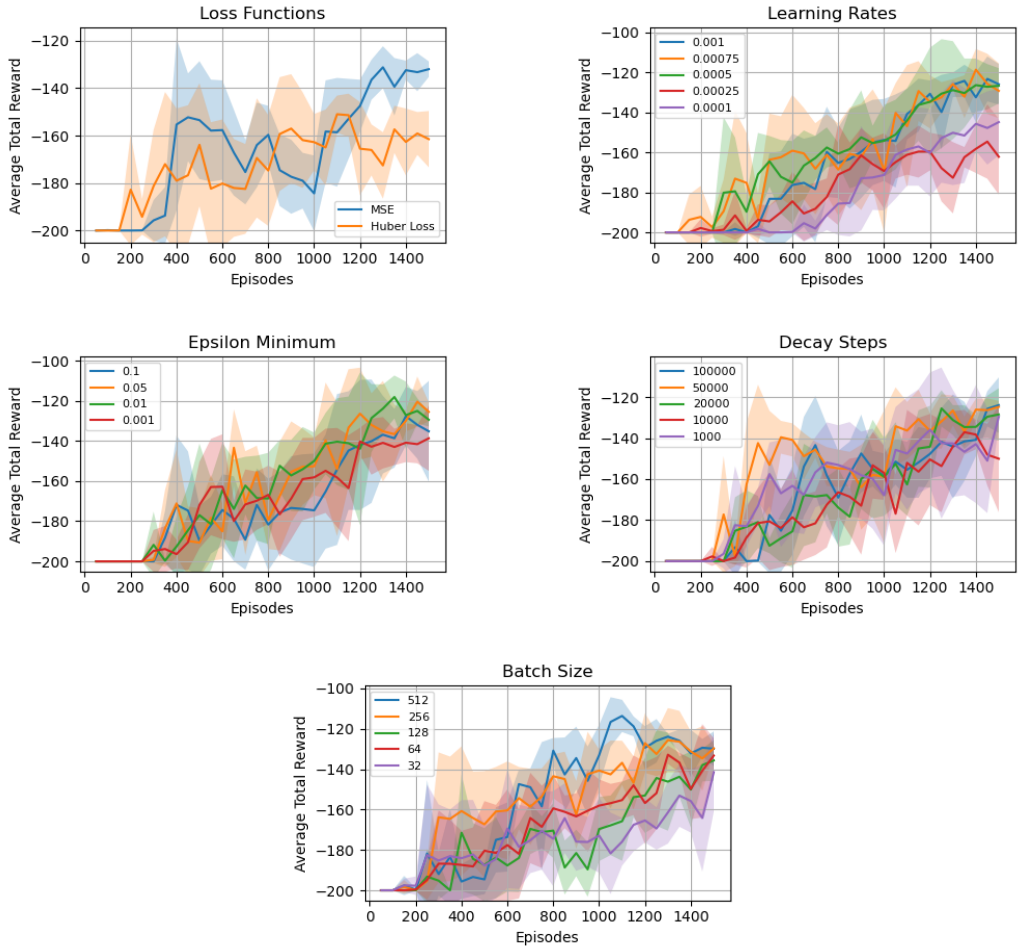
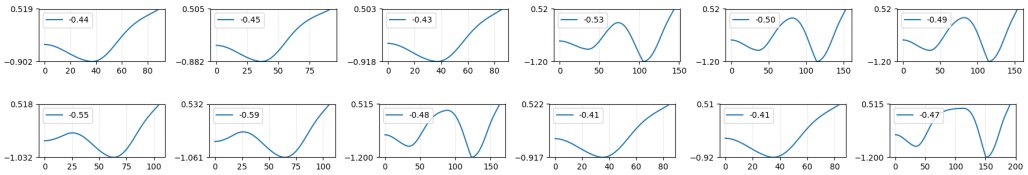
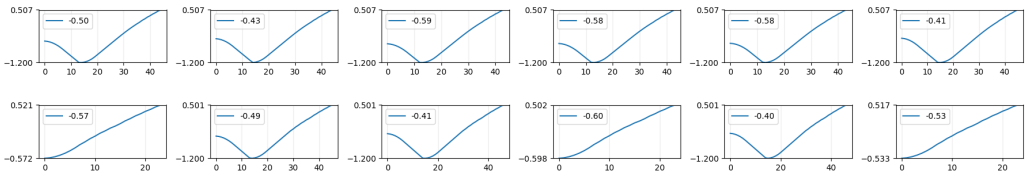


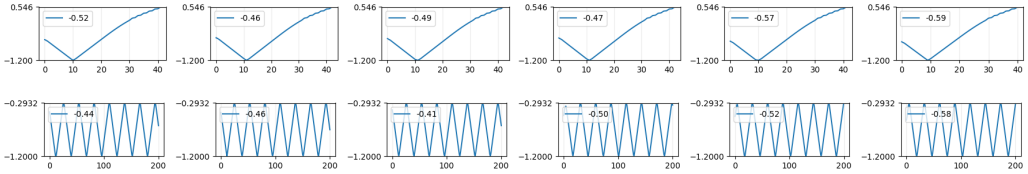
Figure A.2: Graphs showing a variation of tested parameters for the Mountain Car environment. The points reflect sets of 100 validation episodes that were performed every 50 training episodes. The shaded regions represent the standard deviation of the rewards.



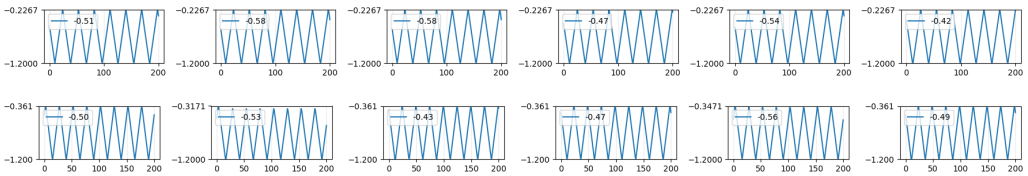
(a) Standard environment



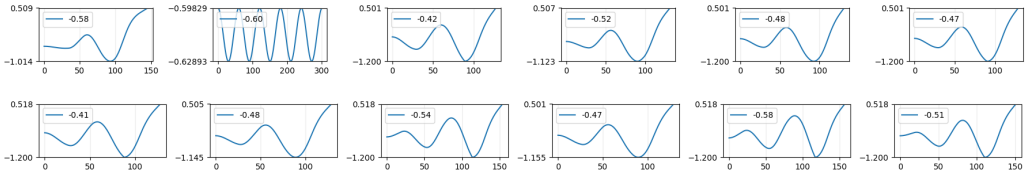
(b) Force = 0.01



(c) Force = 0.05

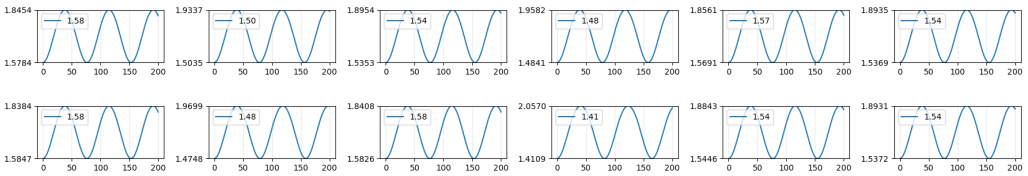


(d) Force = 0.1

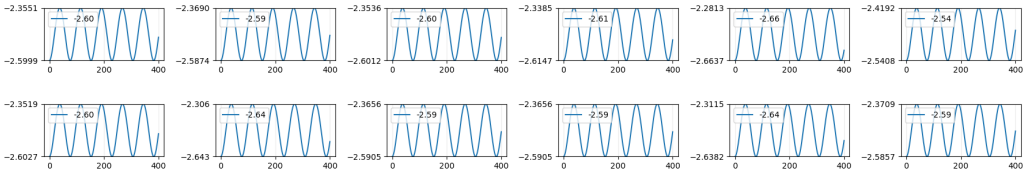


(e) Steeper hill

Figure A.3: Each subfigure shows twelve episodes in different Mountain Car environments. The first row (episodes 1–6) is by the MCV1 agent, and the second row (episodes 7–12) is by the MCV2 agent. Starting positions are labelled in each episode.



(f) Mirrored Terrain



(g) Extended Terrain

Figure A.3: Each subfigure shows twelve episodes in different Mountain Car environments. The first row (episodes 1–6) is by the MCV1 agent, and the second row (episodes 7–12) is by the MCV2 agent. Starting positions are labelled in each episode.

Appendix B

Temperature Regulation Graphs

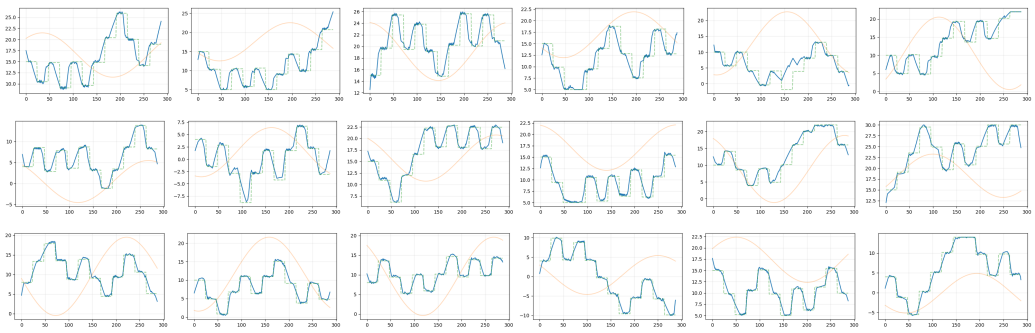


Figure B.1: Eighteen episodes in the standard temperature regulation environment. The first row (episodes 1–6) is by the V4.0 agent, the second row (episodes 7–12) is by the V4.1 agent, and the third row (episodes 13-18) is by the V4.2 agent. The orange, blue, and green lines are the outside, inside, and target temperatures.

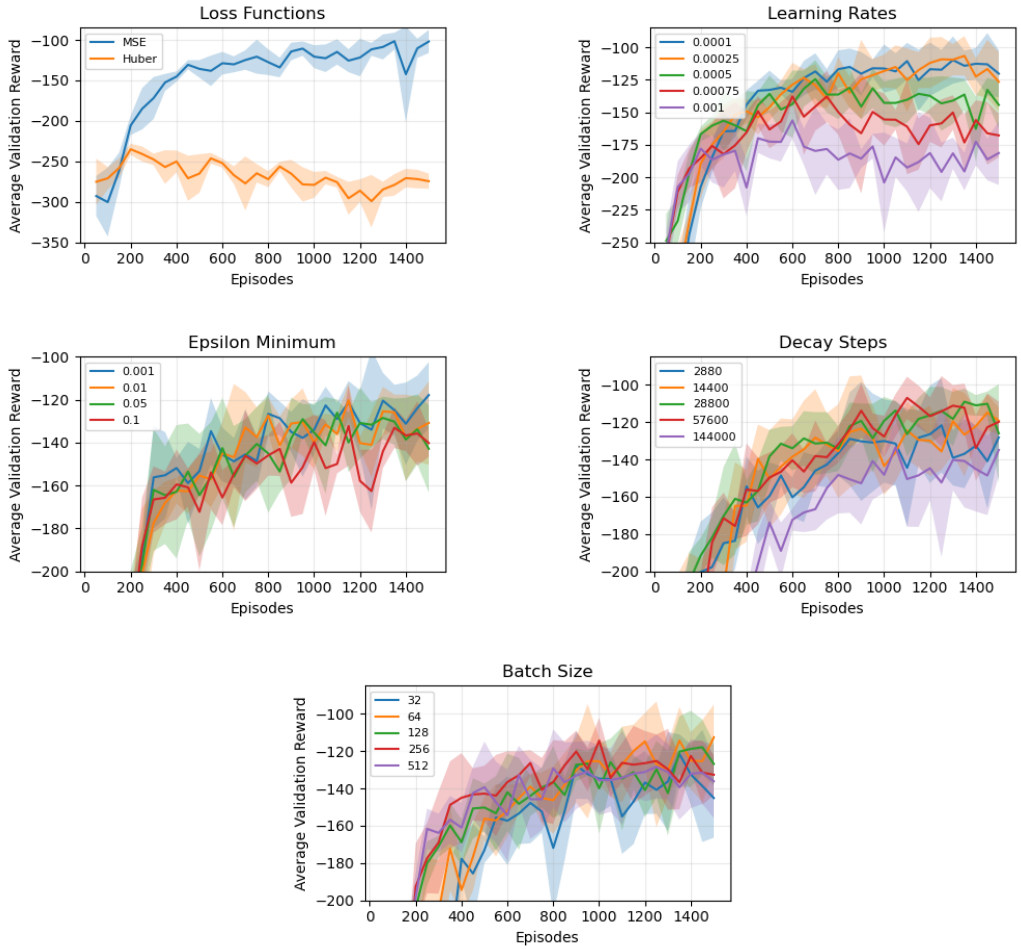
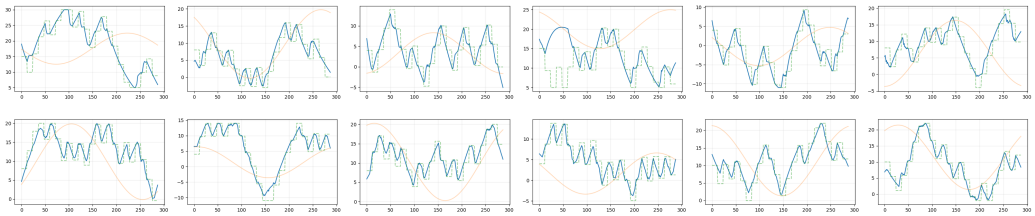
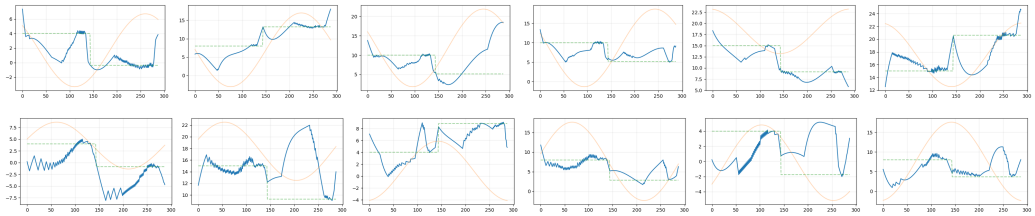


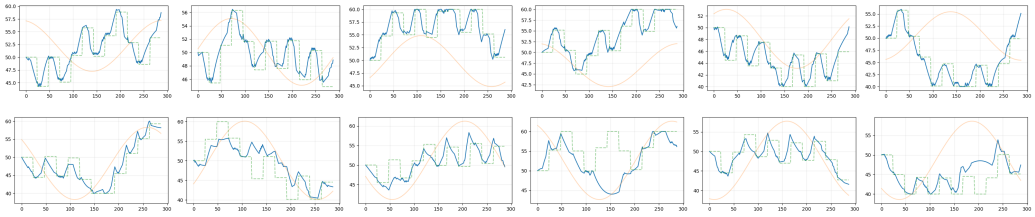
Figure B.2: Graphs showing a variation of tested parameters for the temperature regulation environment. The points reflect sets of 100 validation episodes that were performed every 50 training episodes. The shaded regions represent the standard deviation of the rewards.



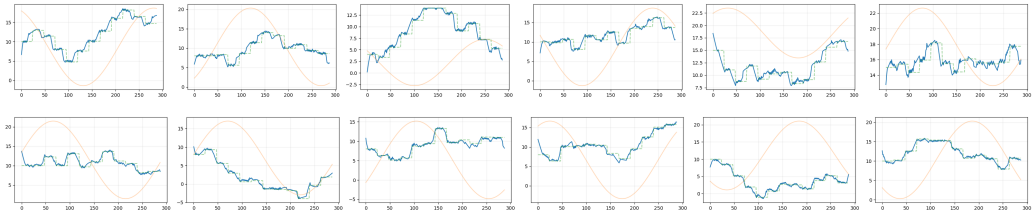
(a) Switch time = 12



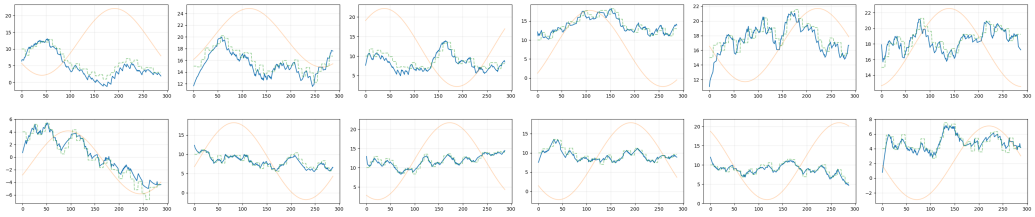
(b) Switch time = 144



(c) Unseen temperatures



(d) Target temperature changes = $[0, 4]$



(e) Switch time = 8 and Target temperature changes = $[0, 2]$

Figure B.3: Each subfigure shows twelve episodes in different temperature regulation environments. The first row (episodes 1–6) is by the V4.0 agent, and the second row (episodes 7–12) is by the V4.1 agent. The orange, blue, and green lines are the outside, inside, and target temperatures.

Bibliography

[1] Boston Dynamics. Starting on the right foot with reinforcement learning. https://bostondynamics.com/blog/starting-on-the-right-foot-with-reinforcement-learning/. Accessed: 19 August 2025.

[2] Farama Foundation. Mountain car. https://gymnasium.farama.org/environments/classic_control/mountain_car/. Accessed: 19 August 2025.

[3] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. arXiv preprint arXiv:1312.5602.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[7] Met Office. Weather and climate summaries. https://www.metoffice.gov.uk/research/climate/maps-and-data/summaries/index. Accessed: 19 August 2025.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016. arXiv preprint arXiv:1511.05952.

[10] David Silver, Thomas Hubert, Julian Schrittwieser, and Demis Hassabis. Alphazero: Shedding new light on chess, shogi, and go. https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/, 2018. Accessed: 19 August 2025.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, Second Edition : An Introduction*. MIT Press, 2018.

[12] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[13] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015. arXiv preprint arXiv:1509.06461.

[14] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016. arXiv preprint arXiv:1511.06581.