REINFORCEMENT LEARNING WITH MOUNTAIN CAR GAME

Jonathn Chang, Queena Deng, Shan Jiang, Oscar O'Brien
Department of Mathematics; University of California, Los Angeles
Los Angeles, CA 90095
{jchang153, qdeng38, shangjiang7, oscaro}@g.ucla.edu

March 10, 2023

ABSTRACT

Reinforcement Learning (RL) has been a field of interest in recent years. In particular, RL is an effective tool for analyzing Markov decision processes. In this paper, we take a deep-dive into RL learning applied to a simple mountain car game with low-dimensional action and state spaces. We utilize this environment to study the effects of various state-of-the-art RL methods, explore the best parameters utilizing an epsilon-greedy policy scheme, and compare the results through graphical visualizations.

Keywords: Reinforcement Learning, Markov decision process, Deep Q-Learning, Gymnasium, TensorFlow

1 Introduction

Reinforcement Learning is one of the main branches of machine learning (along with unsupervised and supervised learning). Unlike the other types of learning, RL assumes a more complex problem structure. Instead of starting with a labeled or unlabeled dataset, RL instead trains an agent whose goal is to maximize a reward scheme by choosing what actions to perform when met with an observation provided by the environment. The agent utilizes a policy to decide how to map a given state to an action, and receives a reward and the next state from the environment. This policy is often chosen to be Epsilon-Greedy, which will be further discussed. The agent trains by updating its mapping of state-action pairs. In theory, the best action to choose is the one that maximizes all future reward for all possible future states, but in practice, we only consider the best possible action in only the next future state, utilizing the Bellman equation to perform updates.

The Mountain Car game is an example of an RL environment in which the observation space is 2-dimensional to describe the position and velocity of a car. The agent is provided three discrete actions at each step: accelerate to the left, don't accelerate, and accelerate to the right. Although the position is along a single dimension, the behavior of the environment (unseen by the agent) is intended to simulate the physics of a car between two mountain peaks; i.e., the environment's generation of states is governed by the gravitational acceleration of a point on a frictionless sinusoidal surface. The car is initially placed somewhere around the valley between the two mountains, and the environment 'stops' or the agent 'wins' when the car reaches a flag on the right peak of the mountain.

Our main approach to training an agent for this game utilizes Deep Q-Learning (DQN). As opposed to vanilla Q-Learning, which trains by updating its Q-table's Q-values (i.e. reward value for given state-action pairs) via Bellman's equation, we instead replace the table with a deep neural network that learns to predict the best possible action, given a state as an input. The loss function is formulated directly from the Bellman equation to allow for propagation and weight updates.

By default, the Mountain Car environment returns -1 as a reward for every observation step in which the agent hasn't yet reached the flag. Unfortunately, using this reward scheme, the car will not learn any beneficial information on how to reach the flag. We instead propose several different reward schemes through initial intuition and derive schemes inspired from methods that have been proved to be successful. This will prove to be challenging, but introduce some fascinating research. In particular, we must translate "reach the flag on top of the hill as quickly as possible" by solely rewarding the car using its current and previous memory of positions and velocities. To reach an invisible goal, we must teach the car to build momentum, something that is far more difficult than, say, the cartpole task (see `https://gymnasium.farama.org/environments/classic_control/cart_pole/`), which can be learned and beaten entirely with immediate rewards.

A rough outline of the paper is as follows. Section 2 will provide more background on RL and Q-Learning, diving into the history and development of these methods. Section 3 will describe the theory behind DQN and Q-Learning and the motivation behind using a deep neural network to learn the RL agent. Section 4 details the implementation of our DQN methods and its various schemes, along with a comparison with vanilla Q-Learning and SARSA. Finally, we conclude with our results, data visualizations, and further research in sections 5 and 6.

2 Background

Q learning was first introduced by Christopher Watkins in his 1989 Ph.D. thesis [6]. He outlines an algorithm for an agent to optimize their actions that could be expanded to machine learning. He notes how research shows animals changing their behavior and "learning" when rewarded or punished for specific actions. He argues that the conditions for optimal learning are worthwhile efficiency, easy exploration, and a short time to learn.

However, Q learning has limitations to discrete input and less complex environments. In 2013, Volodymyr Mnih et al. [3] published the first instance of a reinforcement learning problem successfully solved by optimizing a policy from higher dimensional input. The authors note that the main advantages of their neural network and Q learning variant are reusing experiences in weight updates, reducing updating variance with samples, and avoiding bad local minima with experience replay. Their results in 6 of the 7 games tested were improvements over other models at that time.

A paper similar to ours is "A Brief Study of Deep Reinforcement Learning with Epsilon-Greedy Exploration" by Hariharan N and Paavai Anand G in 2022 [4]. The work examines the effect of exploration during the training process and how that impacts the time for the agent to solve the system. The algorithms are tested on three games, one of which is the mountain car game that we explore in our paper. An epsilon-greedy function is used to balance exploration and exploitation. In the mountain car game, they found their controlled exploration method had a 1.13 performance ratio to the same model with no randomization.

3 Theory

A Markov decision progress (MDP) is at the heart of the mountain car game. For each step during the game, the car has three finite actions to select, given its position ($[-1.2, 0.6]$) and velocity ($[-0.7, 0.7]$) values. In

class, we discussed MDP with finite states and actions. This game clearly has finite actions, but it must also have finite states given finite floating point precision. As a result, an optimal policy must exist.

Furthermore, Fan et al. [2] prove theoretical convergence for Deep Q Neural Networks with rectified linear unit (ReLU) activation in an environment with finite actions and compact states. Their proof requires the Bellman optimality operator can be written as a composition of Holder smooth functions with respect to the state and state-action probability measures are absolutely continuous with respect to the Lebesgue measure.

However, Wang and Ueda [5] comment on Deep Q Learning's success in practice depends on hyperparameter tuning. As a result, models can fail on simple tasks because of a lack of exploration and generalizing on extrapolated data.

4 Methods

4.1 Deep Q-Learning

In this section, we detail the epsilon-greedy policy used for Q-Learning, introduce the Bellman equation, and describe the extension to Deep Q-Learning with formulating a loss function and the structure of the deep neural network. These components are combined in the Deep Q-Network algorithm that we apply to solve the Mountain Car problem. We also describe the various reward functions that we define in order to optimize the agent's performance.

4.1.1 Network Architecture

In the context of reinforcement learning, the goal of an agent is to learn a policy that maximizes its expected cumulative reward. To achieve this goal, the agent must balance the trade-off between exploring the environment to discover new, potentially better actions and exploiting its current knowledge of the environment to maximize its reward. The epsilon-greedy policy provides a way to balance exploration and exploitation by defining a probability distribution over the action space. Specifically, at each time step t, the agent selects an action A_t according to the following probability distribution:

$$P(A_t = a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|}, & \text{if } a = \arg\max_a Q(s_t, a) \\ \frac{\epsilon}{|A|}, & \text{otherwise} \end{cases} \tag{4.1}$$

where $Q(s_t, a)$ is the estimated Q-value of taking action a in state s_t, ϵ is a constant between 0 and 1 that controls the level of exploration, and $|A|$ is the size of the action space. Intuitively, when ϵ is high, the agent is more likely to choose a random action, which encourages it to explore new parts of the environment. As ϵ decreases over time (e.g., by decaying it at each episode), the agent becomes more confident in its Q-value estimates and more likely to choose the action that is currently believed to be the best. This allows the agent to exploit its current knowledge of the environment to maximize its reward.

In order to update and optimize the agent's understanding of which actions to pick in the greedy scenario, we use the Bellman equation, which describes how the Q-values of state-action pairs are updated as the sum of the immediate reward and the discounted future rewards,

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)], \tag{4.2}$$

where s is the current state, a is the current action, r is the immediate reward, s' is the next state, a' is the next action, α is the learning rate, γ is the discount factor, and $\max_{a'} Q(s', a')$ is the maximum Q-value over all possible actions in the next state s'. This equation describes how the Q-value of a given state-action

pair is updated by estimating the optimal future reward assuming the best current action is taken. Notice that the reward r is the main degree of freedom we attempt to manipulate in order to optimize the agent's performance, as described in section 1.

In Deep Q-Learning, we use a deep neural network (DNN) to model the decision process of the greedy scenario. In particular, in a forward pass, the DNN accepts states as inputs, and outputs a Q-value corresponding to each possible action (input states are mapped to action-Q-value pairs). To retrieve the greedy decision, the action with the highest corresponding Q-value is chosen among the outputs of the DNN. In order to train this model, we require 'target' Q-values in a sense to compare to our predicted Q-values. Building off of 4.2, Deep Q-Learning reformulates the Bellman equation into a mean-squared error (MSE) loss function to update the Q-values,

$$\mathcal{L}(s, a, r, s') = (Q(s, a) - (r + \gamma \max_{a'} Q(s', a')))^2, \tag{4.3}$$

where the quantity $r + \gamma \max_{a'} Q(s', a')$ is the Temporal Difference target that we aim to predict. In practice, the quantity $\max_{a'} Q(s', a')$ is obtained by sampling a minibatch of previous transitions, i.e., previous tuples (s, a, r, s'). To optimize equation 4.3, we parameterize $Q(s, a)$ as a DNN, $\phi(s; \theta)$ and train to minimize the loss over θ (weights/biases of the DNN):

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta) = (\phi(s; \theta) - (r + \gamma \max_{a'} \phi(s'; \theta)))^2. \tag{4.4}$$

We make a comment about Huber loss instead of MSE in the appendix, 9.1. For the neural network, we use a simple feed-forward structure, defined recursively as follows:

$$\begin{aligned} \boldsymbol{h}_0 &= \boldsymbol{V}\boldsymbol{x} \\ \boldsymbol{h}_l &= \sigma(\boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l) \\ \phi(s; \theta) &:= \boldsymbol{a}^T \boldsymbol{h}_L \end{aligned} \tag{4.5}$$

where $l = 1, \ldots, L$, $\boldsymbol{V} \in \mathbb{R}^{m \times n}$ (input layer matrix), $\boldsymbol{W}_l \in \mathbb{R}^{m \times m}$ (lth hidden layer matrix), $\boldsymbol{b}_l \in \mathbb{R}^m$ (lth bias vector), and $\boldsymbol{a} \in \mathbb{R}^{d \times m}$ (output layer). The parameters L, m, n, d refer to the depth of hidden layers, the width of hidden layers, the dimension of the state space, and the dimension of the action space, respectively. The activation function σ is flexible, but usually defined as ReLU.

The neural network is trained using gradient descent (in particular, Adam) with a fixed learning rate η. The table of hyperparameters required for training are listed in Table 1. Algorithm 1 details our Deep Q-Network algorithm. Each episode resets the state of the Mountain Car environment, and performs a certain number of maximum steps for which the agent interacts with the environment and the DNN is updated. In particular, if the agent reaches a certain `done` condition (which in the context of this game means when the car reaches the flag), the training loop ends prematurely and the episode ends.

4.1.2 Original and Hyperbolic Reward Functions

The first reward function, "original" (2), consists of two components: a fixed reward of 10 if the car reaches the top of the hill, which incentivizes the agent to reach the goal quickly, and a reward proportional to the square of the car's position, but only if its position is on the right slope (greater than -0.4), which incentivizes the agent to move to the right side of the valley (2a). This reward function makes it easier to reach the flag and earn a high reward, as well as increase its potential energy and gain more kinetic energy, which can result in higher velocity and momentum.

The original reward function for the mountain car problem is simple and incentivizes the car to reach the goal efficiently, which reduces the chance of unexpected behavior and increases training stability. However, the reward function always gives a reward when the car's position is on the right slope, which means that the more steps the agent takes, the higher the total reward it receives. To address this issue, the number of steps taken in each episode can be used to evaluate the agent's performance. Additionally, the reward function only considers the car's position and may not fully capture the significance of the car's velocity in the design of the reward function.

To design a reward function that takes both velocity and position into account, we developed a second test function using a hyperbolic function. The $\tanh$ function, was chosen due to its ability to create a non-linear mapping between input variables and reward values. In this function, the input to the $\tanh$ function is calculated as the absolute value of velocity multiplied by 10 plus the position plus 0.5. By scaling the velocity from the range $[-0.07, 0.07]$ to $[-0.7, 0.7]$ and adding 0.5 to the position, the velocity and position values are mapped to the range $[-0.7, 1.8]$. Then we can apply this as input to the $\tanh$ function. Its algorithm 3.

The $\tanh$ function is a smooth and differentiable function that is beneficial for reinforcement learning. It can capture complex relationships between variables and create a smooth reward landscape, making it easier for the learning algorithm to converge to an optimal policy2b. By using this hyperbolic function, our reward function can more effectively incentivize the car to reach the goal while also taking into account the significance of its velocity.

4.1.3 Plus Velocity and Human Reward Functions

Since we want to explore the training effects of different reward functions, an obvious modification to the original reward function is adding a velocity component. Intuitively, this seems like a crucial piece of information for the model because the car needs to build up its velocity over time to reach the goal.

Similar to the original reward function, the plus velocity reward algorithm 4 gives a large reward for reaching the flag. It differs by giving a fixed reward to the car if the car moved closer to the goal, or the magnitude of its velocity increased. This reward function is still fairly simple and gave mixed results, discussed later.

Since this game is straightforward for a human player, we wanted to the training effects of a "human" reward function 5. Beyond the reward for reaching the flag, the largest fixed reward for the model comes from the car slowing down as it moves up the right side of the hill. The rest of the fixed rewards are the same when the car slows down going up the right side of the hill or increases the magnitude of its velocity going down either side of the hill. This reward scheme is more complex than our others, but it explicitly rewards the correct behavior. Unfortunately, this model had very little success in practice.

4.1.4 Adibyte Reward Function

The final reward function we test is inspired by the user linked in 7. This user formulated the reward function 6 which achieved excellent results; we utilized this reward function as a sort of control to compare and improve other reward functions.

Because the reward is calculated at each step of the agent's training using its current state (position and velocity), the intuition behind this reward function is to reward the agent for whenever the action it chooses causes an increase in velocity, i.e., the velocity becomes more negative if it was already traveling to the left, and the velocity becomes more positive if it was already traveling to the right. Because the agent attempts to maximize reward, this incentivizes the car to choose actions that increase its velocity, so in theory, it should reach the top of the mountain if trained correctly. The reward is also decremented every step to punish the car for doing nothing, or for decelerating against the current direction of travel. Finally, a large reward is awarded when the car reaches the flag in order to attempt to replicate this behavior.

4.2 Q Learning and SARSA

As a comparison to our Q-learning algorithm, we also implemented a SARSA algorithm in the mountain car problem. Unlike the Q-learning algorithm which is off-policy and updates the Q-value using the max Q-value of all possible actions for the next step, SARSA updates the Q-value using the policy. Essentially, the update rule selects the next action using the same policy used to select the current action. In our mountain car problem, the SARSA algorithm calculates the Q-value of the next state using the epsilon-greedy policy. The Q-function for the SARSA algorithm is

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)], \tag{4.6}$$

where a' is now the next action taken by the agent in the next state s', which is derived from the epsilon-greedy policy.

4.2.1 Reward Function

For our Q-learning and SARSA models, the agent gets a reward of -1 at each step if it has not reached the goal and 0 if it has. Thus, the agent will try to achieve the least negative reward. We also terminate the episode at 500 steps in order to shorten training time. To evaluate how well our models perform for Q-learning and SARSA, we primarily utilize average reward over every 100 episodes.

4.2.2 Hyperparameters and Epsilon Decay Functions

We try to optimize our Q-learning and SARSA models through the alteration of different hyperparameters: learning rate α, gamma (or discount) rate γ, and initial epsilon ϵ. Additionally, we focus on the effects of altering the epsilon reduction functions.

Our **original epsilon decay** is a linear decay function that takes into account the number of episodes we train on. After each episode, epsilon is reduced by

$$\frac{\epsilon - \epsilon_{\text{min}}}{N} \tag{4.7}$$

For most of our models, we use a ϵ_{min} of 0, so epsilon may get infinitely small but never reach 0. This version of the epsilon decay function ensures that for any number of episodes we train our models on, epsilon is reduced proportionally.

Similarly, we also test a **simple linear function** that does not account for the number of episodes and instead uses the number of training steps. After each episode, the epsilon is reduced by

$$\frac{\epsilon - \epsilon_{\text{min}}}{M}, \tag{4.8}$$

The linear function decreases epsilon by a constant value at each episode.

Another decay function we attempt is a **stalled linear epsilon decay**, which only implements the linear decay based on episodes after a certain number of episodes. In most of our models, we choose to delay epsilon reduction for the first 300 episodes.

Next, we attempt an **exponential decay function**. After each episode, epsilon is multiplied by a factor of

$$(\frac{0.001}{\epsilon})^{\frac{1}{N}} \tag{4.9}$$

Thus, epsilon decays a lot faster initially.

We also test a **discrete interval decay** where epsilon is reduced by 0.005 every 100 episodes.

5 Results

5.1 Original and Hyperbolic Reward Function

After implementing the original function, I observed that the car successfully reached the goal in just a few episodes. As shown in Figure 3b, the car took only 10 additional episodes to reduce the number of steps required from 400 to 120 after initially reaching the goal. The performance of the car was very stable, with the exception of a single outlier. Additionally, there was a slow downward trend in the number of steps required per episode. The results were outstanding, with a minimum of only 84 steps required. The corresponding phase plot in Figure 3e illustrates that the car only required one back-and-forth movement to reach the goal.

It typically takes more than 10 episodes for the agent to reach the goal for the first time when using the hyperbolic reward function. But, the step plot in Figure 4b shows small oscillations in consecutive episodes. This phenomenon occurs because the $\tanh$ function heavily depends on the initial position of the car, which is a random number in the range [-0.6,-0.4]. When the car starts from the left side of the local minimum, it could gain enough momentum to reach the goal by moving forward, backward, and forward again, as seen in the phase plot4c. However, when the car starts from the right side, it is unable to gain enough momentum to reach the goal in one back-and-forth movement, as opposed to the original reward function. As a result, it takes two back-and-forth movements, resulting in more steps. The mathematics logic causes such difference is that $\tanh$ function slowing down the reward it gives to the car as it approaches the goal, compared to the square function used in the original reward function 2.

It should be noted that the aforementioned results are the best outcomes achieved among all of the training runs. To obtain these results, several hyperparameter adjustments were made, followed by testing of the corresponding results. As mentioned earlier, the decay rate is a crucial hyperparameter for the epsilon-greedy policy that determines the optimal balance between exploration and exploitation of the agent. I tested three different decay rates: 0.997, 0.9995, and 0.9999. From the plot 1, it can be observed that the decay rate of 0.997 was too steep, resulting in limited exploration by the agent. On the other hand, the decay rate of 0.9999 resulted in relatively slow training compared to the optimal decay rate of 0.9995.

Initially, I trained my original reward function model with a learning rate of 0.01, which performed well for the first 60 episodes. However, it then started to oscillate between two peaks of the hill instead of reaching the goal. The plot 3d shows that it takes more steps after that. The reason behind this issue is that as the agent becomes more familiar with the environment and starts to converge towards a policy that works well, a smaller learning rate may cause the agent to become stuck in a local minimum or oscillate between different suboptimal policies. Additionally, the design of the original reward function presents a problem whereby the more steps the agent takes, the more reward it receives, which can cause the agent to oscillate in pursuit of higher rewards. To address this problem, I increased the learning rate to 0.002 which is shown in table 2, allowing the agent to explore more and potentially escape from the oscillation.

Overall, both the original and hyperbolic reward functions perform very well under certain hyperparameters, as shown in tables 2 and 3. The decision space plot for both functions 3 and 4 show that they follow the similar rule of moving left [value:0] on the left side of the hill and vice versa. However, there is a slight difference between them. Specifically, when the agent moving on the left side of hill with high absolute speed, hyperbolic reward function prefer to move right [value:2], as seen in plot 4. This behavior can cause the hyperbolic function to gain different amount of momentum on the left side of the hill compared to original function.

5.2 Plus Velocity and Human Reward Function

One challenge I faced when training these DQN models was poor performance. Especially since training over 100 episodes took approximately 40 minutes, it was frustrating to see a model that felt like a waste of time. I felt that the training would benefit immensely from a reward function that accounted for the car's increased speed rather than just considering its position (original reward function).

The plus velocity reward function was my first attempt at rewarding the car for going faster. After adjusting the hyperparameters 4 many times, the plots 5 show a successful model that reached the flag 90 times out of 100 5b. However, when retraining the model with identical parameters to create the phase 5c decision space 5d plots, the results were much worse since the fewest steps in an episode to reach the goal were 217 compared to 122 before. These vastly different results with identical hyperparameters point to early exploration before too much epsilon decay is extremely important for later exploitation. For further improvement, finding a way to scale the reward rather than a fixed one might improve training results since some of our more successful models used that method to train cars that used fewer steps on average to reach the flag.

The human reward function was my second attempt at making a more comprehensive reward scheme (hyperparameters 5). This model performed extremely badly, which is very apparent in the plots 6 through how many times the training reached the max number of episode steps 6b. Its best episode was reaching the flag in 394 steps, which was the most out of all fully trained models, and its phase plot 6c shows many inefficient oscillations. Clearly, the underlying problem with this method was a reward function that was encouraging the wrong behavior, as the score function shows an upward trend 6a even though the model performance was consistently poor. The decision space 6d clearly displays the problem as the "optimal" policy for a large chunk of the graph is to not accelerate (shown in blue). My takeaway from this attempt is to not overcomplicate the reward function. The model is using the fact that it can stop accelerating and use its velocity to be rewarded for continuing to gain elevation in the short term while hurting its long term goal of reaching the flag by slowing down.

5.3 Adibyte Reward Function

While training the adibyte reward function 6, I noticed large inconsistencies in the car reaching the flag. In particular, with epsilon dictating the probability of the agent choosing random actions (and hence exploring more), if the car managed to reach the flag early in the training due to random chance, the large reward it would obtain would signal to the car to continue this behavior. However, if the car does not manage to reach the flag before the epsilon decays to a small value, not enough exploration will have been done to inform the car that it should attempt to reach the flag. This would lead to the car essentially oscillating around the valley of the two mountains for the entirety of the training. Moreover, the car could reach the flag extremely early due to random chance, but not learn this behavior due to the parameters not being adjusted yet.

The original implementation used an epsilon decay rate of 0.995, which, as displayed in 1, would reach the minimum value of epsilon extremely early in the training process. Therefore, the original implementation heavily relied on random chance in order to train the model. On the other hand, along with trying several other (slower) epsilon decay rates as in 1, my final results were obtained by implementing an additional functionality in the policy, in which epsilon would be reset to 0.9 for each episode that the car doesn't reach the flag, and decreased to 0.3 for each episode that the car does reach the flag (assuming epsilon isn't already lower than this value). This would allow the model to continue exploring if it hasn't reached the flag, but limit exploration and increase exploitation if it has reached the flag. If it happens to luckily reach the flag and the epsilon decreases to 0.3, any future episodes where it doesn't reach the flag would increase it back to 0.9 to continue learning. Essentially, the exploration doesn't stop until the car can consistently reach the flag. Combining this with a slower decay rate, the training becomes more robust and is more consistent in reaching the flag.

Several changes were also made to the reward function to improve training. Because the agent receives no sense of time (steps), I increased the reward at the flag if it reached it more quickly (i.e. if it reaches the flag, the reward received would be inversely proportional to the number of steps it took). To incentivize the car's final push to the flag, I also added more reward to the car based on how close it is to the flag.

The final results of these changes can be viewed in 7. The values of hyperparameters used for training are in 6. In particular, 7a and 7b describe the agent's cumulative reward and steps taken over the course of the 100 episode training period. Although in the first 10 episodes, the steps taken is at 600 (i.e. the car never reaches the flag), every single episode past that mark reaches the flag in around 100 steps, with a minimum of 91. The phase plot 7c displays the performance of a model that took 110 steps during training, where we can see the car moves slightly up the right side of the hill, then accelerates all the way up the left hill, then uses this momentum to reach the flag. In total, the valley is only crossed three times. We can also see the decision space 7d of the optimal model, which shows that the model consistently chooses 0 (accelerate left) when on the left hill, and 2 (accelerate right) when on the right hill, with very little values of 1 (indecision, or no acceleration).

5.4 Q-Learning and SARSA

The initial models of Q-learning and SARSA utilize a learning rate of 0.2, a gamma of 0.9, and an initial epsilon of 0.8. Training on different numbers of episodes, we were able to determine that SARSA tends to perform better during training, but both algorithms converge to approximately the same final reward value, as seen in figure 8 However, an issue these models share is that they are very slow to reach the goal for the first time. Even after completing the game successfully for the first time, the model takes many episodes before it's able to learn how to build momentum. When rendering the models, I found that the car spends many episodes oscillating between the two hills, and at times going very far up the right hill but not crossing the goal.

In the mountain car problem, it is important to explore the environment thoroughly in order to find an optimal policy. While Q-learning is more likely to get stuck in local optima, SARSA avoids that issue because it is more likely to explore the environment systematically. This higher average reward over the episodes for SARSA is the result of the on-policy training. At each step, because SARSA is more conservative in its updates, it can avoid overestimating Q-values, which leads to more stable performance during training, which we see in our plots.

Training Q-learning models on different learning rates, models get the best results with lower learning rates around 0.1. Similarly, models with higher gammas around 0.95 achieve better rewards. For initial epsilons, lower initial epsilons tend to provide better results. The results of these hyperparameter tuning attempts are shown in figure 9

However, the alteration of these hyperparameters never seemed to improve the models significantly, so I looked to develop the models by changing the epsilon reduction functions. Training Q-learning models with the different epsilon reduction functions we have discussed above, the linear, exponential, and discrete interval functions achieve the best results, with linear and exponential converging the fastest, as seen in figure 10a. This is most likely due to the fact that the linear and exponential functions decay epsilon much faster than the other three functions. Figure 10b shows similar results are achieved when SARSA models are trained with the different epsilon reduction functions. The linear and exponential reduction functions still tend to converge the fastest.

As a final best model comparison, we run Q-learning and SARSA with both linear and exponential epsilon decay functions for 5000 episodes. Learning rate is set to 0.1, gamma to 0.95, and initial epsilon to 0.1. In figure 10c, the initial Q-learning and SARSA models are also plotted for performance comparison. As

expected, the models with linear and exponential reductions converge faster and achieve better results. These models still converge to approximately the same final average reward though. Of these four models, the exponential Q-learning model had the shortest run with 85 steps. However, this run was achieved on episode 1900 of 5000 episodes, which indicates that this may be a result of random "luck" when exploring rather than a consequence of the model learning and converging. Thus, it's hard to state with confidence that among the linear and exponential decay function models, there is a best one.

6 Conclusion and Further Research

Considering the results from all of our testing, simpler reward functions tended to have the most consistent success in the straightforward mountain car game. With Deep Q Learning, the original reward function, which only considered the car's position, had the best single-episode performance. The hyperbolic and adibyte functions were close behind with similarly basic reward schemes. Moreover, Q Learning and SARSA, which have the drawbacks of requiring discrete input and more iterations, had performance on par with the neural network models. The human reward function was clearly over-engineered and is a great example of trying to force the agent to learn a certain way, which the agent does not quite understand. Overall, we were surprised by how variable our results were for such a simple game across the different reward functions and adjustments in hyperparameters.

To further demonstrate the advantage of Deep Q-Learning would require further experimentation, particularly with the best-performing reward functions (original, adibyte, hyperbolic). With limited computing resources and time throughout the course of this project, it would be worthwhile to test each of these functions with more episodes and larger DNNs. There's also a lot of potential for future research in policy gradients, genetic algorithms, and evolution strategies applied to the mountain car problem.

7 Data Availability

The python environment defining the game mechanics for the mountain car game comes from the gymnasium package at `https://gymnasium.farama.org/`. See [1] for more information about the deprecated gym package.

The general structure of the Deep Q Learning code came from `https://github.com/adibyte95/Mountain_car-OpenAI-GYM` and the Q-learning code from `https://gist.github.com/gkhayes/3d154e0505e31d6367be22ed3da2e955`. The original reward function came from `https://github.com/shivaverma/OpenAIGym/tree/master/mountain-car`.

8 Contribution Statements

Development of question / hypothesis: Each of us helped discuss and conceive the project idea.

Data research: Oscar found gymnasium python package and skeleton code.

Literature review: Jonathn and Shan researched the Deep Q-Learning algorithm. Queena researched the Q-Learning and SARSA algorithms. Oscar researched the Deep Q-Learning background and theory.

Analysis strategy: Everyone analyzed their respective methods.

Code writing/analysis/review: Jonathn tested the adibyte reward function and helped implement Deep Q-Learning algorithm, training loop, and data saving functionality. Queena implemented and tested the Q-Learning and SARSA algorithms, and improved these models with epsilon decay functions. Shan tested the original reward function and created and tested the Hyperbolic reward function. Oscar created and tested

the plus velocity and human reward functions, wrote code to save videos of models, and fixed code to work with the new gymnasium package rather than the deprecated gym package.

Work planning and organization: We all planned and organized meetings. We always met up as a group. We met 6 times outside of class.

Making or improving plots and visualization tools: Queena made Q Learning and SARSA plots. Shan made original and hyperbolic reward visualizations. Oscar made score, step, and phase plots. Jonathn made the decision space plots.

Improving teamwork and collaboration: We all collaborated as a team through text messages and in-person meetings.

Testing code and procedures: We tested, criticized, and modified each others' code every day.

Writing report: Jonathn wrote the abstract and introduction sections, the network architecture section, and the DQN algorithm and parameter tables. Queena wrote Q-Learning and SARSA method and results sections. Shan wrote the epsilon greedy policy and MSE Huber loss in the network architecture sections. Oscar wrote the background and theory sections.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[2] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning, 2019.

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

[4] Hariharan N and Paavai Anand G. A brief study of deep reinforcement learning with epsilon-greedy exploration, 2022.

[5] Zhikang T. Wang and Masahito Ueda. Convergent and efficient deep q network algorithm, 2021.

[6] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

9 Appendix

One issue with the MSE loss function 4.3 is that it is sensitive to outliers, which can cause the algorithm to overestimate the Q-values in certain states. This can lead to instability and slower convergence of the algorithm. To overcome this issue, the Huber loss function is a better choice for the Mountain Car problem. The Huber loss function is defined as the following:

$$\mathcal{L}(s,a,r,s') = \begin{cases} \frac{1}{2}(Q(s,a) - (r + \gamma \max_{a'} Q(s',a')))^2, & |Q(s,a) - (r + \gamma \max_{a'} Q(s',a'))| \leq \delta \\ \delta(|Q(s,a) - (r + \gamma \max_{a'} Q(s',a'))| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{9.1}$$

where δ is a hyperparameter that controls the threshold at which the Huber loss function switches from quadratic to linear. Though we alternate between MSE and Huber loss in our experiments, we continue to refer to 4.4 as it is more concise.

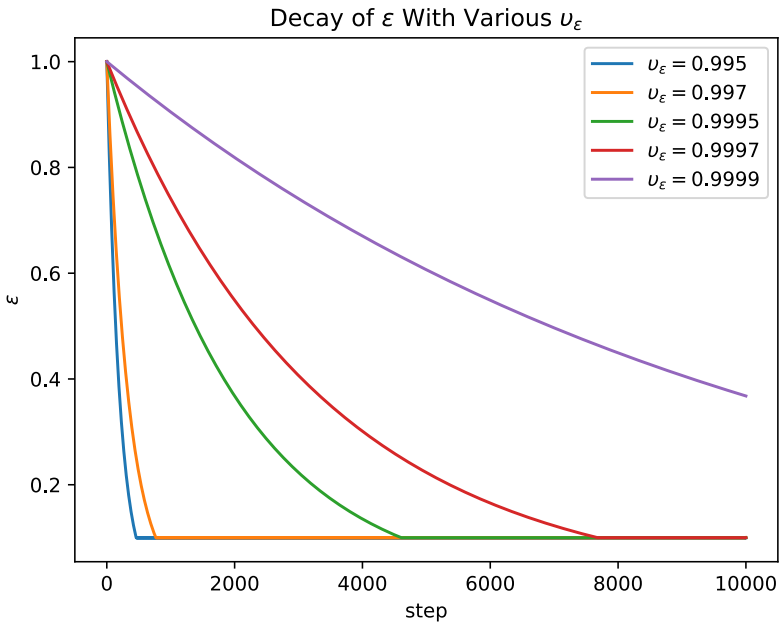


Figure 1: Comparison of Epsilon Decay Rates

Algorithm 1 Deep Q-Network Algorithm

Require: An instance of the Mountain Car environment, `env`, and `reward_type`
Ensure: The parameters θ^* that solve 4.4.

Set hyperparameters listed in 1
Initialize $\phi(s;\theta)$ with random parameters.
for e in $1,\ldots,N$ **do**
    Initialize s from `env`
    for i in $1,\ldots,M$ **do**
        Choose a from s using policy in ...
        Observe r, s' from `env` given a
        procedure UPDATE ϕ
            Update $r \leftarrow$ reward generated by `reward_type`
            Store transition (s, a, r, s') into the replay buffer and randomly sample a B-size minibatch from the replay buffer
            Compute gradient of loss 4.4 with respect to θ using the minibatch and γ, and denote it g
            Update $\phi \leftarrow \phi - \eta g$
            Update $\epsilon \leftarrow \epsilon \cdot \upsilon_\epsilon$ if $\epsilon \geq \epsilon_{\text{min}}$
        end procedure
        Update $s \leftarrow s'$
        if `done` **then**
            Break out of the inner loop
        end if
    end for
end for

Parameter	Definition	Parameter	Definition
ϵ	initial epsilon value	n	dimension of state space and input layer
ϵ_{min}	minimum value of epsilon	d	dimension of action space and output layer
υ_ϵ	decay rate for epsilon	η	learning rate of ϕ
γ	discount factor	N	number of episodes
L	depth of hidden layers in ϕ	M	maximum number of steps per episode
m	width of hidden layers in ϕ	B	batch size of minibatch

Table 1: General Hyperparameters for Training

Algorithm 2 Original Reward Function

$r \leftarrow 0$
if car reaches flag **then**
    $r \leftarrow r + 10$
end if
if car is on right slope **then**
    $r \leftarrow r + (1 + \text{position of car})^2$
end if
return r

Algorithm 3 Hyperbolic Function

$r \leftarrow 0$
if car reaches flag **then**
    $r \leftarrow r + 100$
else
    $r \leftarrow r - 0.2 + 2 * (\tanh(|\text{velocity of car} \times 10| + \text{position of car} + 0.5))$
end if
return r

Algorithm 4 Plus Velocity Reward Function

$r \leftarrow 0$
if car reaches flag **then**
    $r \leftarrow r + 10$
end if
if position of car is further right or magnitude of velocity is greater **then**
    $r \leftarrow r + 1.5$
end if
$r \leftarrow r - 1$
return r

Algorithm 5 Human Reward Function

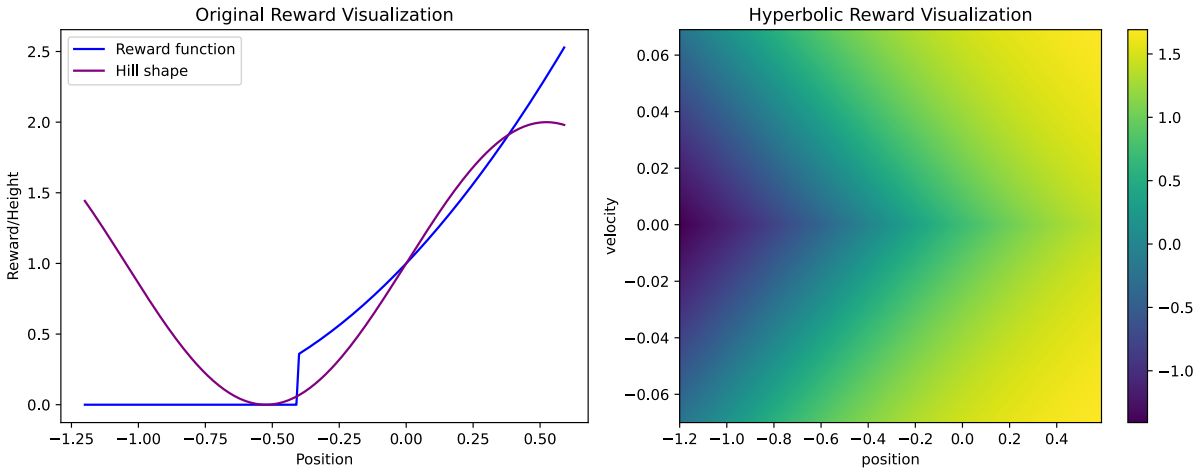
$r \leftarrow 0$
if car reaches flag **then**
    $r \leftarrow r + 10$
end if
if car is moving right and slowing down on right slope **then**
    $r \leftarrow r + 2$
end if
if car is moving left and slowing down on left slope **then**
    $r \leftarrow r + 1$
end if
if car is moving right and speeding up on left slope **then**
    $r \leftarrow r + 1$
end if
if car is moving left and speeding up on right slope **then**
    $r \leftarrow r + 1$
end if
$r \leftarrow r - 1$
return r

Algorithm 6 Adibyte Reward Function

$r \leftarrow 0$
if car reaches flag **then**
    $r \leftarrow r + 10$
end if
if car's velocity is positive and increasing **then**
    $r \leftarrow r + 1.5$
end if
if car's velocity is negative and decreasing **then**
    $r \leftarrow r + 1.5$
end if
$r \leftarrow r - 1$
return r

Parameter	Value	Parameter	Value
ϵ	1.0	n	2
ϵ_{min}	.01	d	3
υ_ϵ	.9995	η	.002
γ	.95	N	100
L	2	M	600
m	32	B	64

Table 2: Hyperparameters for Original Model



(a) Original Reward Function and Hill Shape

(b) Hyperbolic Reward Function

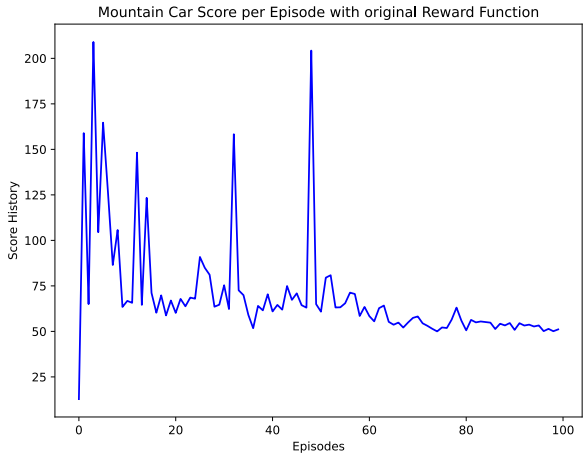
Figure 2: Reward Visualizations

Parameter	**Value**	**Parameter**	**Value**
ϵ	1.0	n	2
ϵ_{min}	.01	d	3
υ_ϵ	.9995	η	.001
γ	.95	N	100
L	2	M	600
m	32	B	64

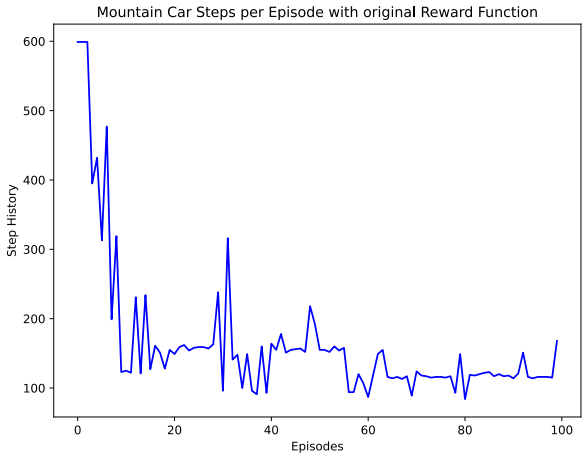
Table 3: Hyperparameters for Hyperbolic Model

Parameter	**Value**	**Parameter**	**Value**
ϵ	1.0	n	2
ϵ_{min}	.01	d	3
υ_ϵ	.9995	η	.001
γ	.95	N	100
L	2	M	600
m	32	B	64

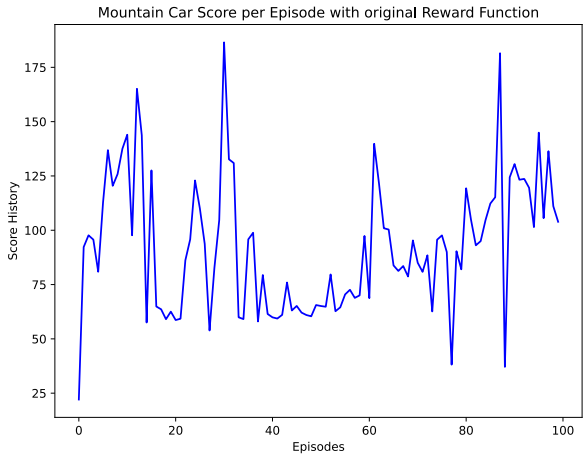
Table 4: Hyperparameters for Plus Velocity Model



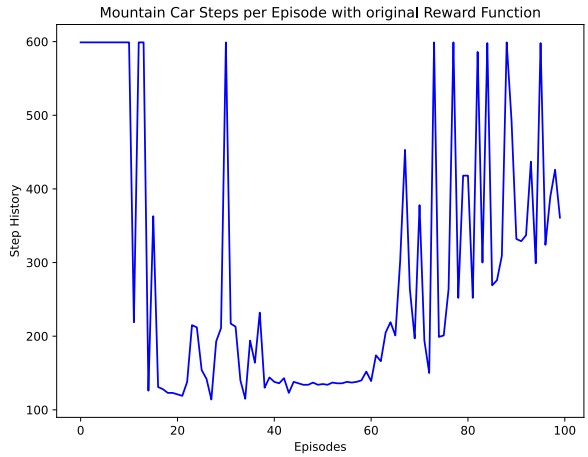
(a) Score vs Episode Plot



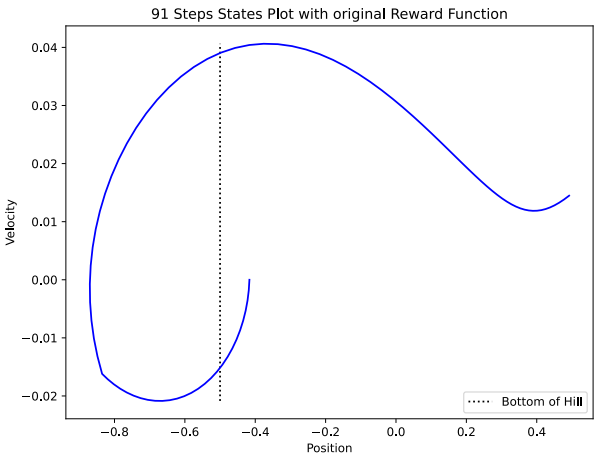
(b) Steps vs Episode Plot



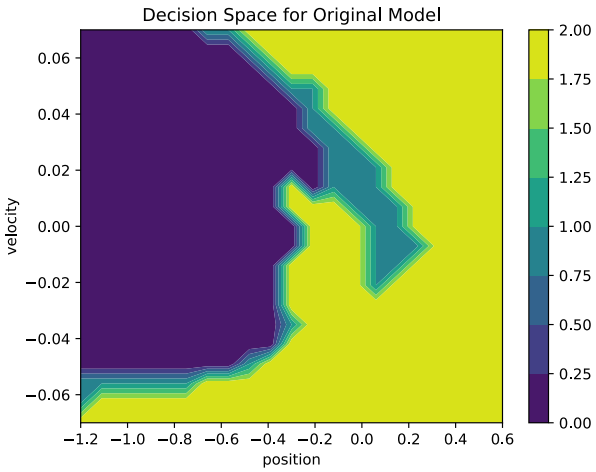
(c) Score vs Episode Plot(learning rate=0.001)



(d) Steps vs Episode Plot(learning rate=0.001)

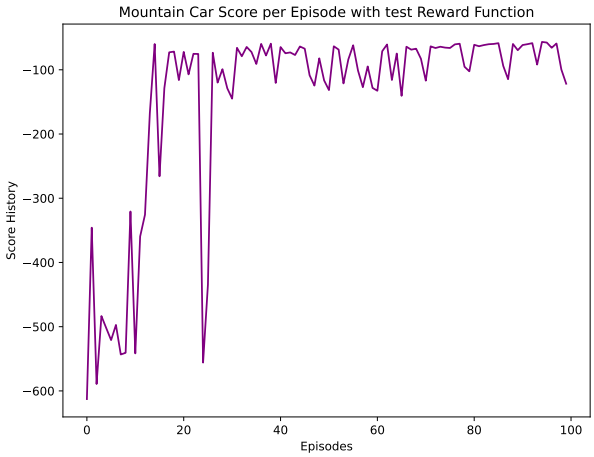


(e) Phase Plot

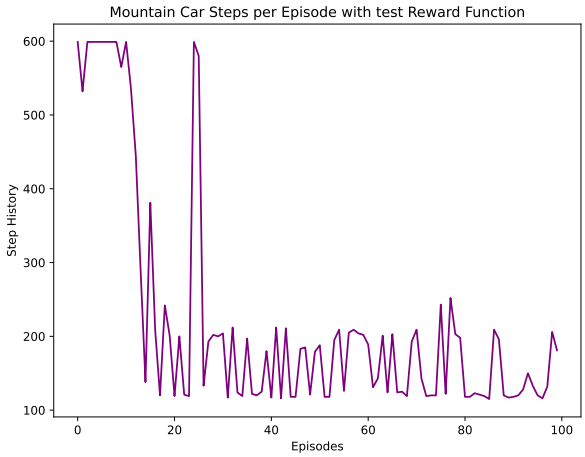


(f) Decision Space Plot

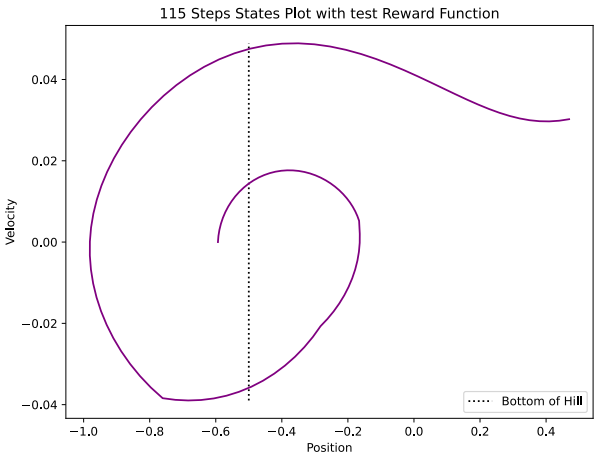
Figure 3: Result Visualizations for Original Reward Function



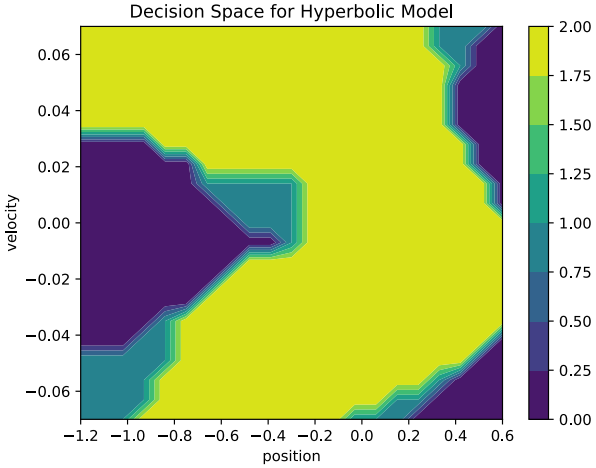
(a) Score vs Episode Plot



(b) Steps vs Episode Plot



(c) Phase Plot

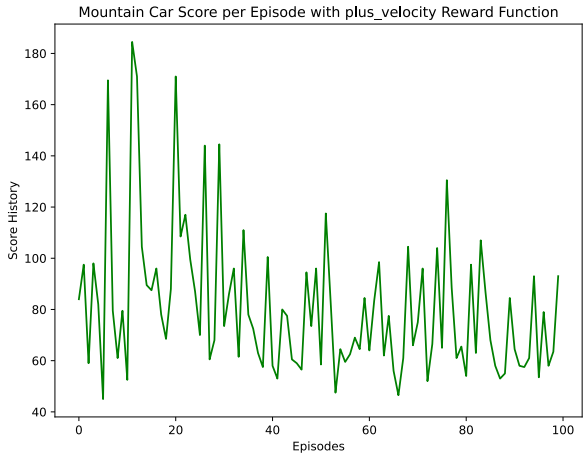


(d) Decision Space Plot

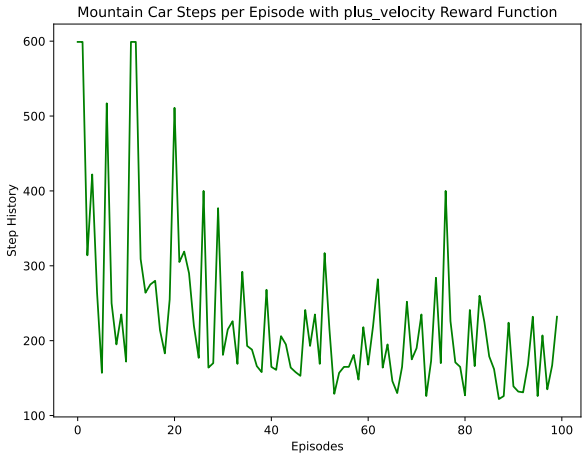
Figure 4: Result Visualizations for Hyperbolic Reward Function

Parameter	**Value**	**Parameter**	**Value**
ϵ	1.0	n	2
ϵ_{min}	.01	d	3
υ_ϵ	.9995	η	.002
γ	.95	N	100
L	2	M	600
m	32	B	64

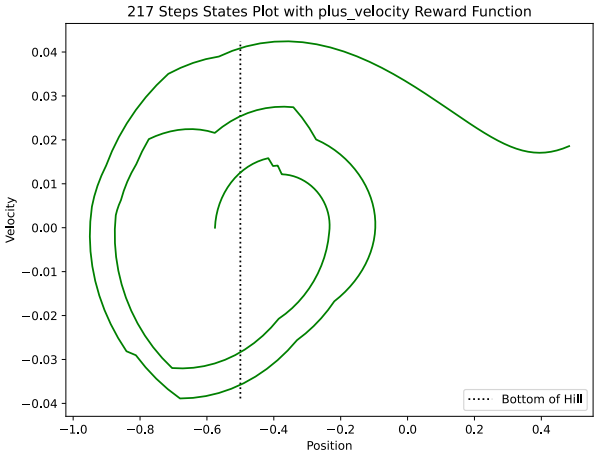
Table 5: Hyperparameters for Human Model



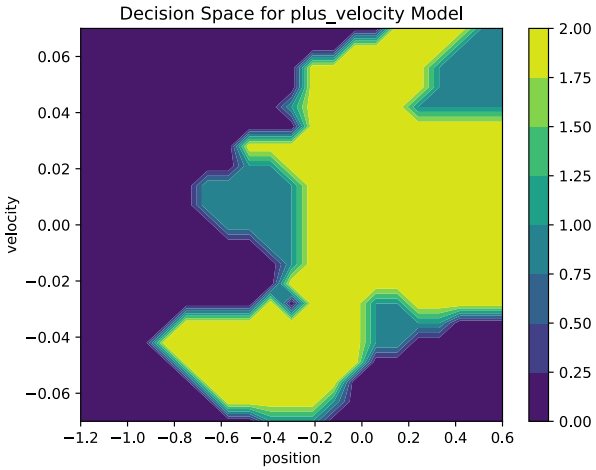
(a) Score vs Episode Plot



(b) Steps vs Episode Plot



(c) Phase Plot

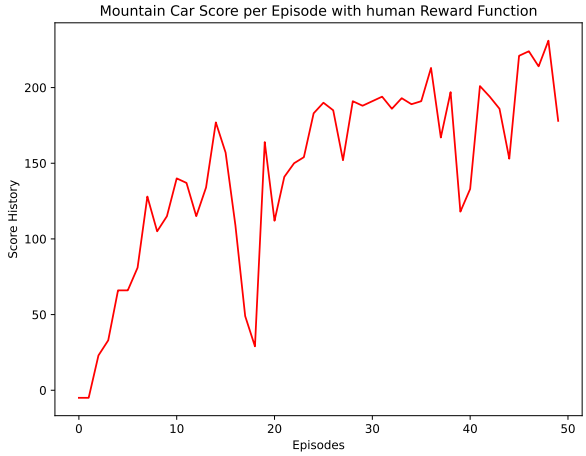


(d) Decision Space Plot

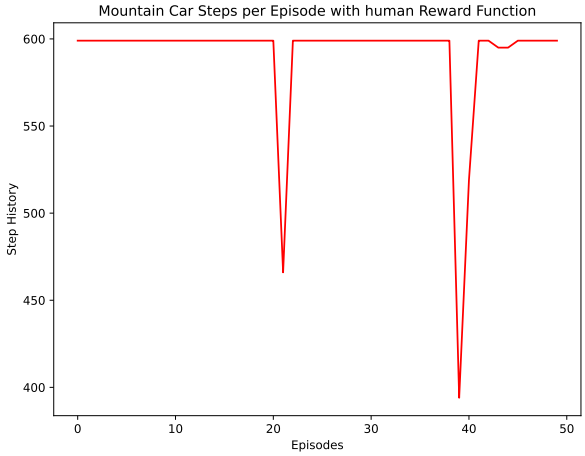
Figure 5: Result Visualizations for plus_velocity Reward Function

Parameter	**Value**	**Parameter**	**Value**
ϵ	1.0	n	2
ϵ_{min}	.01	d	3
υ_ϵ	.9995	η	.002
γ	.95	N	100
L	2	M	600
m	32	B	64

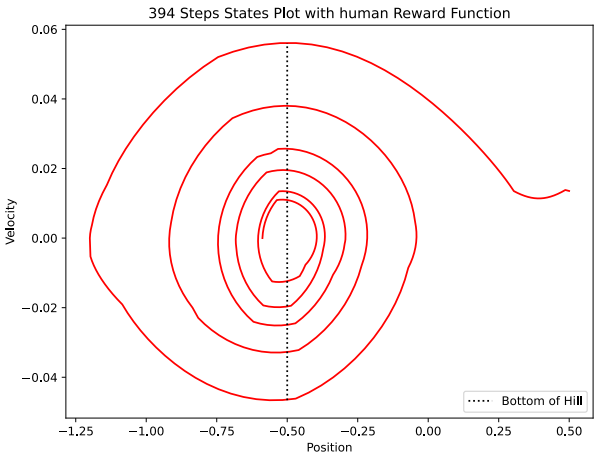
Table 6: Hyperparameters for Adibyte Model



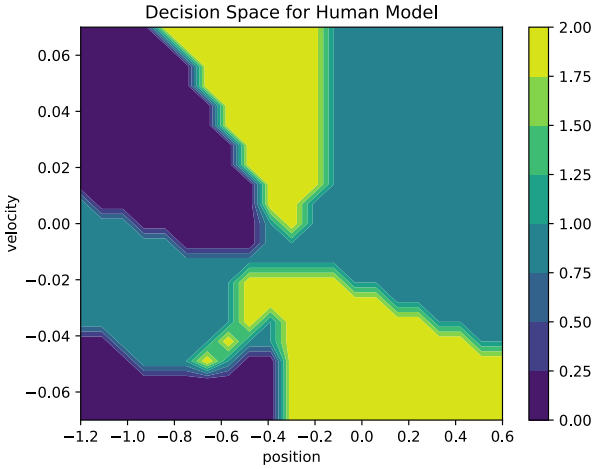
(a) Score vs Episode Plot



(b) Steps vs Episode Plot

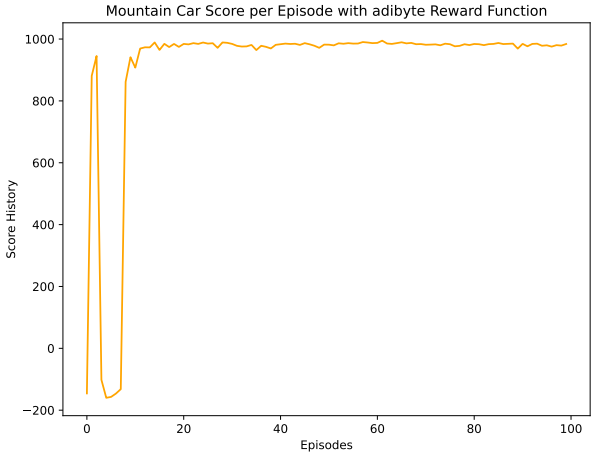


(c) Phase Plot

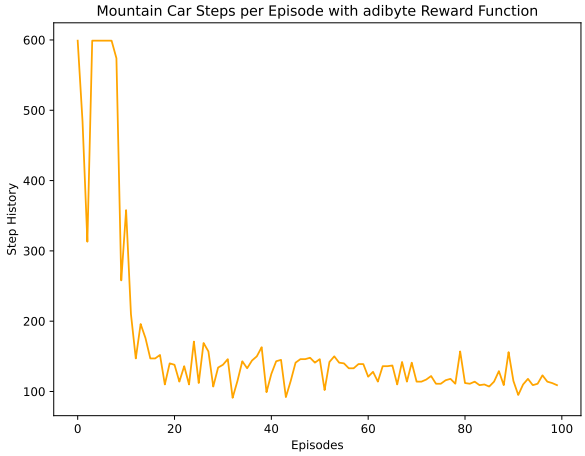


(d) Decision Space Plot

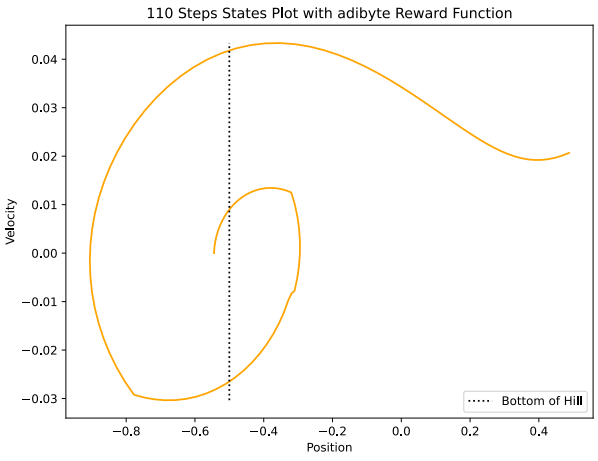
Figure 6: Result Visualizations for Human Reward Function



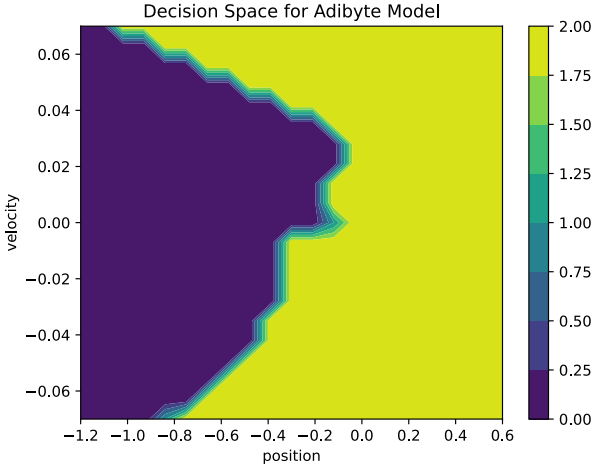
(a) Score vs Episode Plot



(b) Steps vs Episode Plot



(c) Phase Plot



(d) Decision Space Plot

Figure 7: Result Visualizations for Adibyte Reward Function

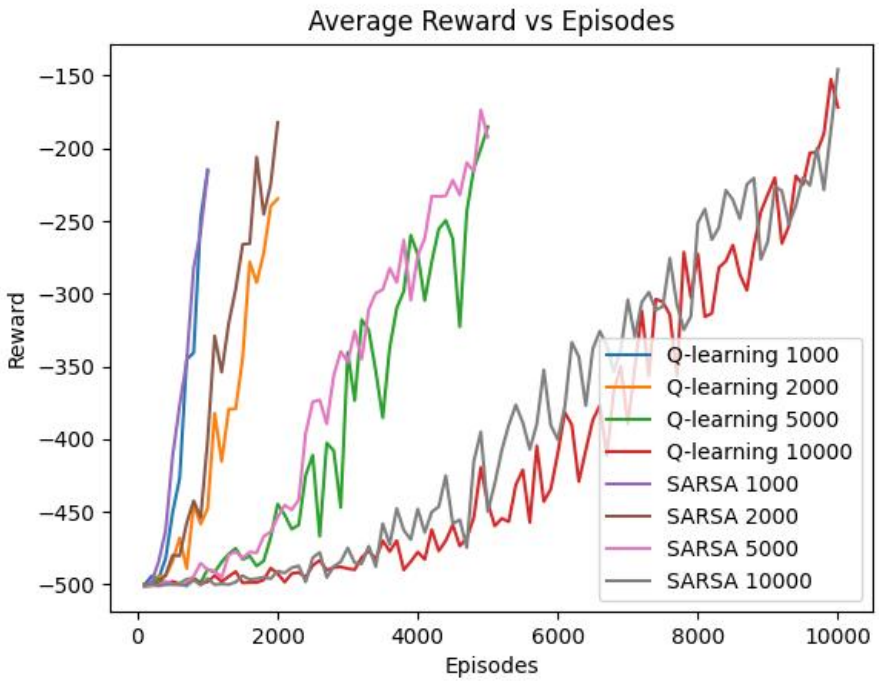
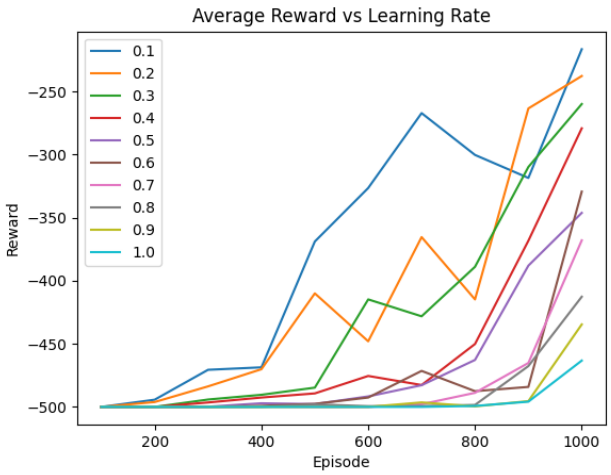
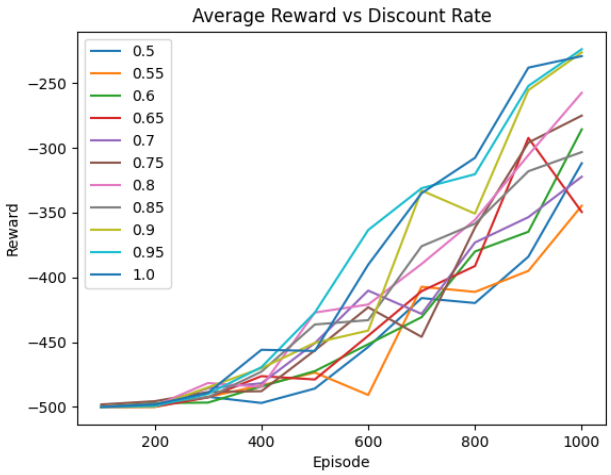


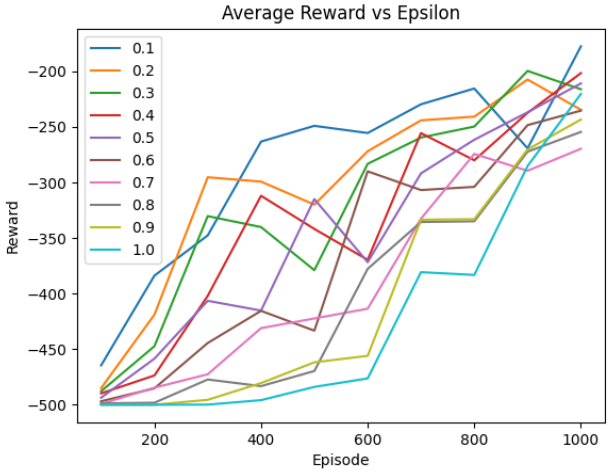
Figure 8: Q-Learning vs SARSA



(a) Learning Rates

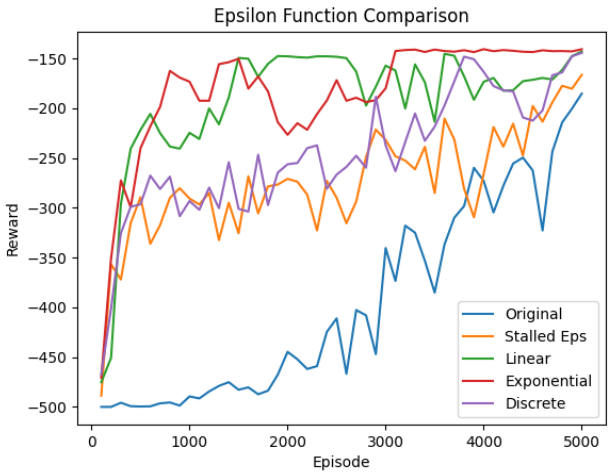


(b) Discount Rates

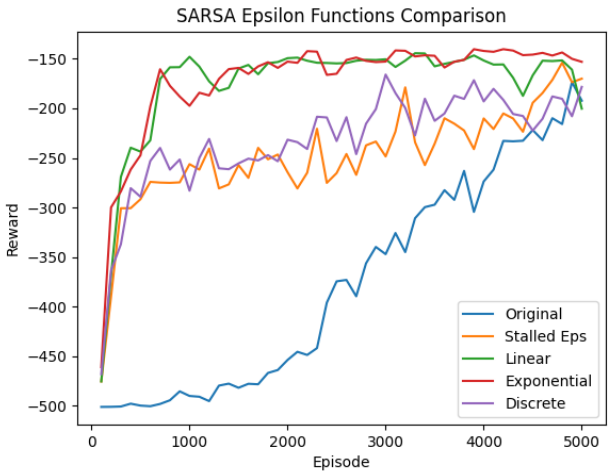


(c) Initial Epsilon

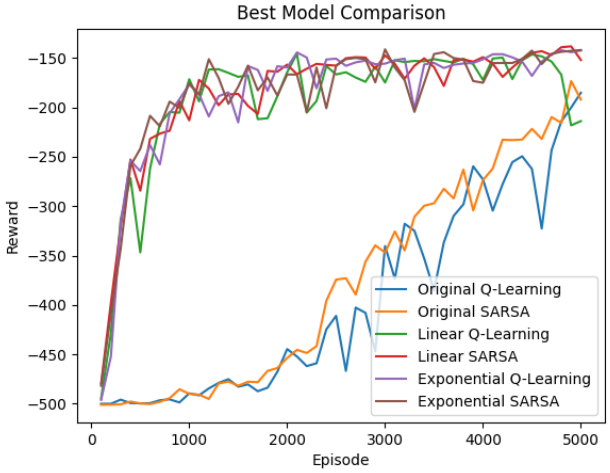
Figure 9: Hyperparameters



(a) Q-Learning with Epsilon Functions



(b) SARSA with Epsilon Functions



(c) Best Models Comparison

Figure 10: Epsilon Decay Function Comparisons