Going to the Gym: Game-playing AI using RL-Q

Morgan Lunn

1 Introduction

Reinforcement learning [1] has been successfully been applied to a wide variety of problems, ranging from robot control [12] to games such as checkers and Go [10]. This method of machine learning is particularly effective in environments where no model of the environment is available, or if there is such a model, but no analytic solution is known.

In this experiment we shall take a pair of games from the open resource 'openAI Gym' and investigate the question 'how do reward functions affect an agent's ability to learn?'.

In the background, we will discuss the theoretical underpinnings of the model. In the Methods section, we will explain how exactly we implemented those methods, in addition to describing the source of our games. In Results, we will first explain how the games worked, and then we will present the results obtained by our agent. Additionally, if we implemented solutions to problems we encountered, we will shortly discuss them here. The Discussion will contain a review of the results, a more elaborate discussion of the difficulties we encountered, and the possible solutions to them.

2 Background

In this experiment we used a model using reinforcement learning (RL) and Q-learning (QL) [3]. Learning from Delayed Rewards] to allow our agent to learn from feedback it receives from the environment in the form of 'rewards'. In this case, the rewards are simply represented by numbers that are received by the agent. The core of this method is the Bellman equation, which describes how to update Q:

The letter 'Q' here refers to the 'quality' of the actions taken [6]. As can be seen, there are multiple hyperparameters which can be tweaked in order to influence the efficacy and speed of learning. Our choice of parameters is given in the following section.

3 Methods

We used a type of reinforcement learning algorithm with Q-learning. This type of learning is especially appropriate for game-playing AI, because it does not require a model of the environment (it is model-free), because it only relies on

reward feedback for reinforcement. Here we used the ϵ-greedy method to trade off exploration versus exploitation. Although some methods can outperform ϵ-greedy [11], we chose the latter for its simplicity and transparency.

Initially, we only used a very simple model, without experience replay. The pseudocode of this simple model is as follows:

if argument == 'train' **then**
    for run in RUNS **do** init state, done, count
        while not done **do**
            if $random() > \epsilon$ **then**: Action = argmax qTable(state)
            else
                $Action \leftarrow random(action\_space)$
            end if
        end while

        $New\_state, reward, done = step(action)$
        $max\_futureQ = max(qTable(new\_state)$
        $currentQ = qTable(state)$
    end for
end if
$newQ \leftarrow (1 - learning\_rate) * currentQ + learning\_rate * (reward + discount * max\_futureQ)$
$qTable(state) + action \leftarrow newQ$
$State \leftarrow new\_state$
decay(epsilon)
save(qTable)
if a **then**rgument == 'play' load(qTable)
    for r **do**un in RUNS Init state, done, count
        while n **do**ot done
            if $random() > \epsilon$ **then**
                $Action \leftarrow$ arg max $qTable(state)$
            else
                $Action \leftarrow random(action\_space)$
            end if
        end while
        $New\_state, reward, done \leftarrow step(action)$
    end for
end if

The hyperparameters were as follows:

$$\begin{aligned} \text{LEARNING\_RATE} &= 0.01 \\ \text{DISCOUNT} &= 0.95 \\ \text{RUNS} &= 3000 \\ \text{TEST\_RUNS} &= 100 \\ \text{GAME\_OVER\_PENALTY} &= -400 \\ \text{START\_REWARD} &= 200 \end{aligned}$$

The source for our games was the open resource 'Gym' by openAI [8]. In particular, we used games from the 'Classic Control' library, which contains

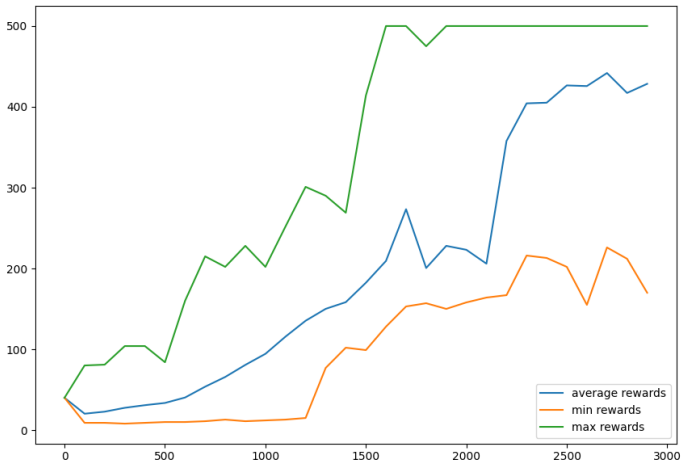

Figure 1: Number of steps per run in the CartPole game over 3000 runs

exercises for agents learning about movement, such as robot arm movement. The two games we chose from this library were 'CartPole' and 'MountainCar'.

4 Results

The results obtained by our agent were starkly different between games. One game, CartPole, was rather simply learned by the agent and yielded satisfying results. However, the other game we tried, MountainCar, did not enjoy the same treatment. We will summarise the working of the games, and then we will discuss the results.

4.1 CartPole

CartPole, henceforth CP, is a game wherein the agent must balance a pole upon a cart for as long as possible. It consists of two actions, namely left and right, and four state variables, namely cart velocity, position, pole angle and pole angular velocity. Additionally, it returns a variable 'done', which returns True when the game is over (and False otherwise). The failstate of this game is reached when either: the pole angle is greater than ± 12 degrees; the cart position is greater than ± 2.4, or the episode length is greater than 500.

The simple model was able to achieve a maximum of 500 steps during training, which is the maximum, in around 1400 episodes. The playing agent is then able to use the qTable from training to achieve an average of 420 on a validation set of 100 extra episodes, which can be seen in .

This result was obtained after applying two simple changes to the reward structure of the game. Firstly, since 'done' returns True both when the game is successfully completed and when the agent has failed, we therefore define a counter variable to count the amount of episodes that have transpired. We

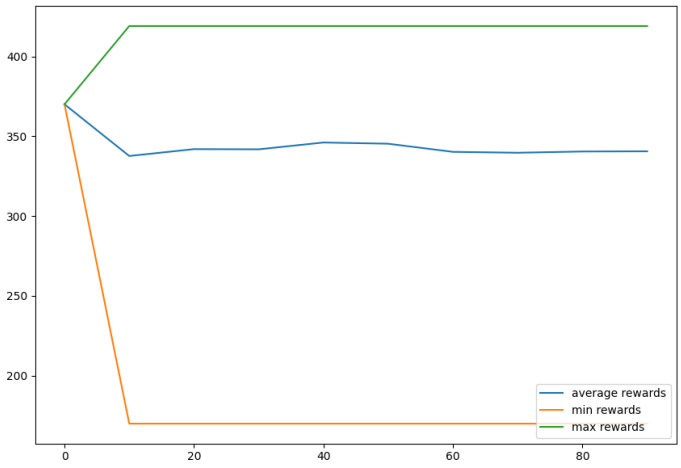

Figure 2: Validation of the CartPole game over 100 runs

then define a new failure state as 'done' being True, while the counter is also less than 500. If this is the case, we administer a large negative reward to the agent, so that it is strongly disincentivised from taking actions which lead to this outcome.

In effect, this redefinition of the failure state reshapes our reward function. This is called 'reward shaping', and it can help our agent learn faster in certain situations [5][7][4].

4.2 MountainCar

However, for MC, the situation was rather different. The agent was unable to successfully finish the game. That is, it was not able to find a way to reach the flag in time during the 3000 runs we set as standard at the start of the experiment. When given the opportunity to play for 10,000 runs, it was still not able to find the right way to control the car so as to reach the goal. It is likely that the reason was the reward system, which is practically binary. This created a strong class imbalance, wherein there was no success state providing positive feedback which the agent could use to modulate its behaviour. Simply put, it was continually failing, but unsure how much, and therefore unsure how to change it and begin succeeding.

Solving this would require a slightly more complicated modification to the reward system, and some deeper knowledge of the inner workings of the game. The documentation for the game [9] tells us that the success state is achieved when the position of the car equals or exceeds the value 0.5. Thus, we can simply say that the car receives precisely this value as reward. Because 0.5 is the highest value this variable can take on, maximising this value will also maximise success in the game.

Indeed, this alteration proved fruitful. It is clear from Figure 3 that the

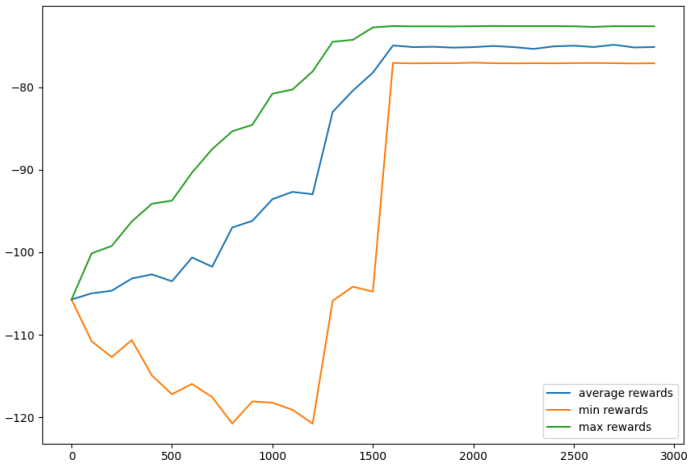

Figure 3: Rewards in the MountainCar game over 3000 runs

agent is learning behaviour which increases its reward, and because we aligned the reward exactly with the success state, this corresponds to learning how to better play the game. Note that the numbers on the vertical axis do not necessarily represent anything in particular; we can just view it as the outcome of the reward function. In Figure 4, the rewards are given by the total distance from the centre over 200 episodes.

Another possibility is to take the squared distance between the flag and the car, but we did not attempt this solution. It's clear that there is something which causes the score to hit the ceiling around -72 (again, because the units are more or less arbitrary, this does not represent anything in particular). Checking the rendered environment, so we can see the agent play in real-time, reveals that the agent is trying not to move away from the flag. However, in order to win the game, the agent must use momentum in order to scale the mountain, which means it needs to accept some less optimal values in order to increase its eventual reward.

A possible solution might be to let the reward be given by the closest the agent has come to the flag thus far, and to give it a high reward if it beats this score, while decaying the reward if it does not beat it.

Unfortunately, even this is not sufficient to persuade the agent to perform better at the game: at around the 1500th run, when the randomness parameter becomes zero, the high score stops being improved upon, and the agent seems to get stuck.

Another possibility [2], is to shape the reward by providing the lowest reward in the situation where the car is in the middle of the valley, so that deviations either way will provide an improvement. Again, however, no solution is found by the agent. The aforementioned source also mentions the reason this is problematic: if no winning solution is ever found during play, the agent cannot properly learn to play the game.

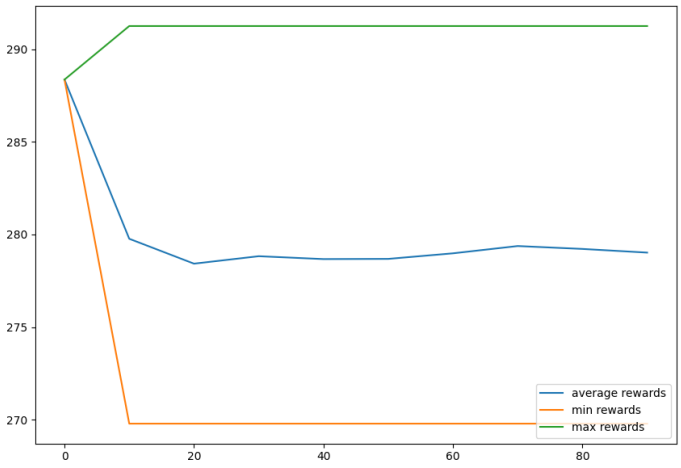

Figure 4: Validation of the MountainCar game over 100 runs

5 Discussion

Rewards are extremely important in reinforcement learning. A large part of the difficulty of this experiment was the implementation of rewards, and in general, the implementation of failstates. The two games we chose to let our agent play, CartPole (CP) and MountainCar (MC), were chosen because they are polar opposites in a sense: CartPole's goal is to stay upright for as long as possible; in MountainCar the aim is to reach the flag as quickly as possible. Since 'running out the clock' in CP is the success state, but this is the failure state in MC, the strategy needs to be very different. In addition, CP requires quick button-pressing, and MC requires the opposite: extended presses of the same button.

For this reason, it was likely that the same reward function would not yield optimal results for both games, and this is indeed what we found. A stronger agent, which builds its own model of the environment, would likely perform better at these tasks. It's possible to reward-shape for every task, but this is clearly not desirable for very large tasks, or tasks which carry greater uncertainty over the rewards. Additionally, human bias about what are good moves may hamper learning.

References

[1] Vincent François-Lavet et al. "An Introduction to Deep Reinforcement Learning". In: *CoRR* abs/1811.12560 (2018). arXiv: 1811.12560. URL: http://arxiv.org/abs/1811.12560.

[2] Genevieve Hayes. *Getting started with reinforcement learning and Open Ai Gym*. Dec. 2019. URL: https://towardsdatascience.com/getting-started-with-reinforcement-learning-and-open-ai-gym-c289aca874f.

[3] Watkins Christopher John Cornish Hellaby. "Learning from delayed rewards". PhD thesis.

[4] Yujing Hu et al. *Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping*. 2020. DOI: 10.48550/ARXIV.2011.02669. URL: https://arxiv.org/abs/2011.02669.

[5] Adam Daniel Laud. "Theory and Application of Reward Shaping in Reinforcement Learning". AAI3130966. PhD thesis. USA, 2004.

[6] Tambet Matiisen. Dec. 2015. URL: https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/.

[7] Andrew Y. Ng, Daishi Harada, and Stuart Russell. "Policy invariance under reward transformations: Theory and application to reward shaping". In: *In Proceedings of the Sixteenth International Conference on Machine Learning*. Morgan Kaufmann, 1999, pp. 278–287.

[8] OpenAI. *A toolkit for developing and comparing reinforcement learning algorithms*. URL: https://gym.openai.com/.

[9] OpenAI. *A toolkit for developing and comparing reinforcement learning algorithms*. URL: https://gym.openai.com/envs/MountainCar-v0/.

[10] David Silver et al. "Mastering the game of go with deep neural networks and Tree Search". In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[11] Michel Tokic and Günther Palm. "Value-difference based exploration: Adaptive control between epsilon-greedy and Softmax". In: *KI 2011: Advances in Artificial Intelligence* (2011), pp. 335–346. DOI: 10.1007/978-3-642-24455-1_33.

[12] Zhaoming Xie et al. "Allsteps: Curriculum-driven learning of Stepping Stone Skills". In: *Computer Graphics Forum* 39.8 (2020), pp. 213–224. DOI: 10.1111/cgf.14115.