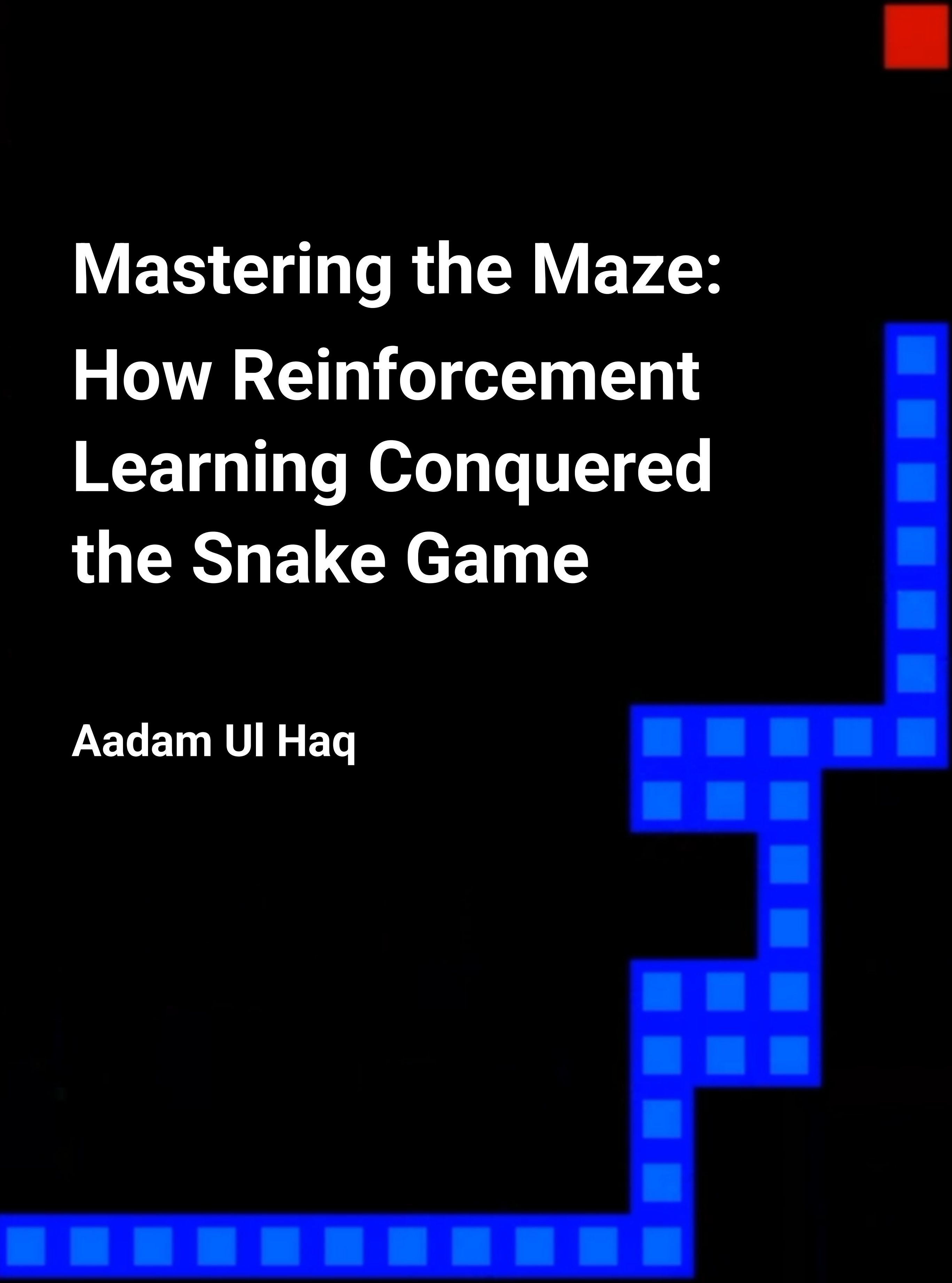
Mastering the Maze: How Reinforcement Learning Conquered the Snake Game

Aadam Ul Haq



Artificial Intelligence (AI) has become a buzzword over the last few years. Breakthroughs such as ChatGPT have propelled discussions that Machine Learning and Artificial Intelligence will emerge in every sector of society.

One branch of Machine Learning in particular, called Reinforcement Learning (RL), has been at the forefront. Its framework is the basis for which ChatGPT runs. Without human intervention, an agent can seemingly learn how to solve tasks by itself, with applications in robotics, self-driving vehicles and healthcare. Its framework is built on animal learning and Pavlovian Conditioning.

On May 11th 1997, World Chess Champion Garry Kasparov, to this day considered one of the best players of all time, conceded the game to IBM's DeepBlue supercomputer. Over 20 years later, DeepMind and Google pioneered an agent, AlphaGo, to become the best Go player, crushing countless World Champions. This begs the question: is AI gradually replacing us?

We will investigate Reinforcement Learning, with the aim of answering three key questions. How easy is it to apply and code using RL techniques? What are the key factors that influence the success of RL algorithms? And most importantly, how well does the RL algorithm perform?

SNAKE GAME: NAVIGATING THE DIGITAL TERRAIN

The Snake Game is a classic and simple game in which you control a snake in the four cardinal directions in order to eat randomly placed fruit. For every piece of fruit consumed, the snake grows in length. The aim is to grow as long as possible, whilst avoiding colliding with the walls of the game and colliding with itself.

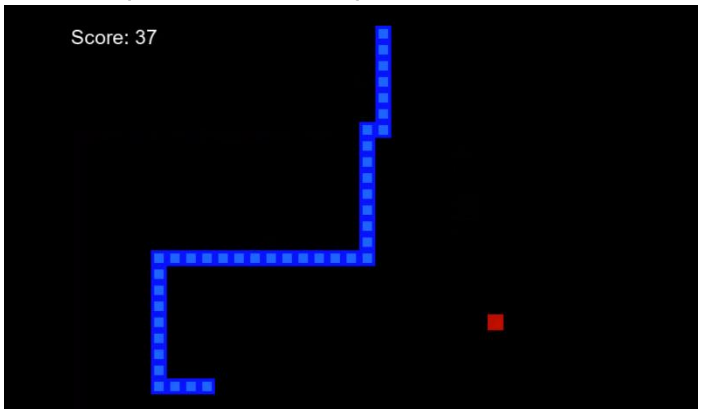


Figure 1: Image taken from the Snake Game. The snake is in blue and the fruit in red.

Although it may be simple, it is the perfect game to test and experiment RL techniques, particularly for beginners. The goal is clearly defined, with simple actions and visual feedback to see how the model performs. Researchers often use simpler video games to test algorithms, specifically taking games from the Atari 2600 console, like Pong (the old line and dot tennis game) and Space Invaders.

The three components that build the framework of RL are states, actions and reward. The states represent the current situation of the environment and encapsulate all relevant information for decision making. The actions, rather intuitively, are the choices the agent can make to move from one state to another. The reward (which can be a punishment too!) is the feedback received by the agent after taking an action.

For simplicity, we will code the RL agent as if it is looking from the position of the snake. In our case, we have 11 states that represent immediate dangers, the relative position of the fruit and the current direction of travel. Our actions are turn left, right or continue straight, and our reward will be set to +10 if we get a piece of fruit and -10 if the agent gets a game over. We will also make the game restart if the snake takes too long to eat the fruit to save time in training.

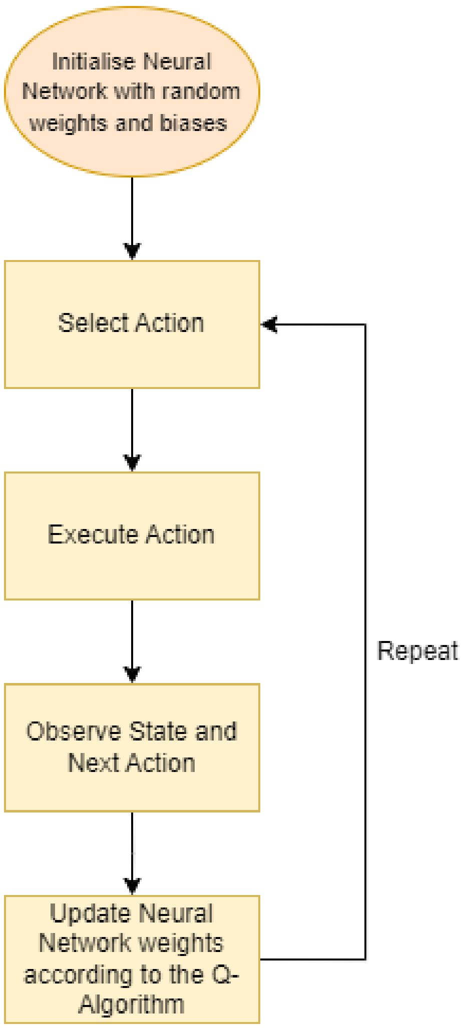
With the groundwork in place, let's investigate the algorithm at the heart of our snake agent.

THE 'BRAIN' BEHIND THE OPERATION

The agent learns what to do based on one simple algorithm, called the Q-Algorithm. We use a variant called the Deep Q-Learning (DQL) Algorithm so we can do calculations using neural networks. The 'Q' is just representing the maximum total reward and is calculated using a simple update formula that balances immediate reward with an amount of any future reward we can obtain.

After we calculate this term at each step, a loss function sees how the new Q-values compare to the old values. This is then used to update the neural network. We then repeat this process the entire time the game runs.

This deceptively simple process is all that is used to create the agent. Created by DeepMind in 2015, the same architecture is used in landing spacecraft, creating intelligent robots and algorithmic trading. Let's see how well the algorithm worked in our particular scenario.



Initialise Neural Network with random weights and biases
Select Action
Execute Action
Observe State and Next Action
Update Neural Network weights according to the Q-Algorithm
Repeat

Figure 2: Deep Q Learning Flowchart

UNVEILING THE POTENTIAL

Amazingly, the agent not only quickly understood its goal, but constantly improved itself to achieve it. A curve showing the rolling average of performance can be found below.

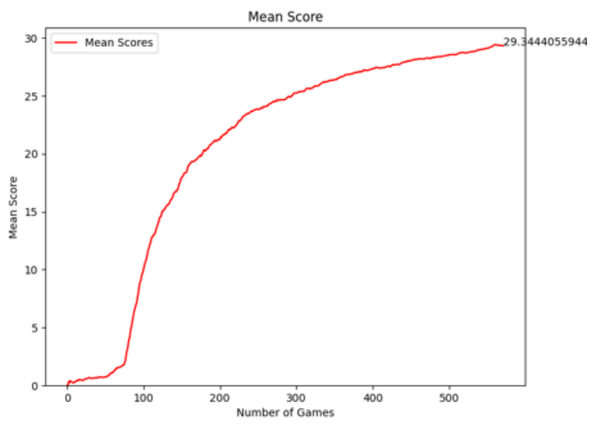


Figure 3: Performance of agent in the Snake Game

In this particular training, the agent achieved a maximum score of 70, a feat that only few humans could attain. This demonstrates good control over the snake, efficient navigation and shows the agent possesses a solid understanding of the game's mechanics.

Amazingly, the agent not only quickly understood its goal, but constantly improved itself to achieve it.

One particular feature of the graph to pay attention to is the rapid increase after 80 games. This shows the boundary between the two phases of learning: exploration and exploitation. Early in the game, random moves occur so the snake can discover what strategies are beneficial. Once it has learned, it enters an exploitation phase in which it capitalises on prior knowledge to achieve a high score.

EXPLORING BOUNDARIES AND BEYOND

Now we have coded an agent for ourselves, we return to our original questions.

Firstly, we notice some limitations of our model. We can see the performance gradually plateaus. Whilst observing the agent running, it was noticed that the snake traps itself as it gets larger. This is due to the way in which the agent was coded. The agent can only see immediate dangers, rather than the whole picture as we would view it. Consequently, it can trap itself in the long term, whilst attempting to benefit in the short term. If the snake were coded a different way, this issue could be mitigated.

The snake was relatively easy to code. The entire agent is built around one algorithm so the simplicity of the model helps beginners in particular at attempting to create their own agent. One challenging aspect however was hyperparameter tuning. These are parameters that affect the learning process. Values must be chosen for these constants, which can heavily influence the performance of the model. Trial and error must be used to locate the best possible values, which may take a while.

Overall, the agent performed extremely well and surpassed all expectations. The seemingly simple algorithm learned the best strategies for the game remarkably quickly, taking at most 20 minutes.

The DQL framework is perhaps the most popular and influential algorithm in Reinforcement Learning. The versatility and performance of the process has created a large impact, motivating researchers to discover more Deep Reinforcement Learning algorithms. The integration of Deep Learning and Neural Networks has enabled scalability and customisation of the process, and allows high-dimensional state spaces to be handled efficiently.

Other RL frameworks also exist, specific to different challenges. These give rise to more impressive applications from ChatGPT to nuclear fusion research. The future of AI looks bright, in large part due to Reinforcement Learning.