

# **Q-Learning**

Peter (Petr) Ladur

56445760

# Chapter 1

## Introduction

The aim of this project is to explore Q-Learning in the context of tic-tac-toe and other applications using python. Theoretical Q-Table was generated using the “minimax” algorithm, and was compared to Q-Tables generated through training against both, random and perfect opponent. Results of agents trained on different policies playing against each other were recorded. How hyperparameters ( $\alpha, \tau$ ) affect training results was investigated and an optimal function for hyperparameters was proposed. Deep-Q-Learning was investigated in the context of different games in the “Open-AI gymnasium” (*I know that this is unlikely but if I finish the report and still have some time left over, I will try playing around with another application of Deep Q-Learning such as the stock market*).

# Chapter 2

## Background

### 2.1 Q-Table

Q-Table is a table which contains expected outcomes after every possible action in a particular state. In the context of tic-tac-toe the state is the game position at that particular moment, and actions are all the empty squares. As seen in table 2.1, for that particular state there are two actions that lead to a win and three actions which lead to a draw (assuming perfect play).

	(1, 1)	(1, 2)	(1, 3)	(2, 1)	(2, 2)	(2, 3)	(3, 1)	(3, 2)	(3, 3)
- - o									
- x -		1	1			0	0	0	
x o -									

Table 2.1: Two actions lead to a win, three actions lead to a draw assuming perfect play from both sides

### 2.2 Q-Learning

In the context of Q-Learning, the Q-Table can be filled up by playing games and updating the values of actions taken during that game according to the Bellman equation.

$$Q(s_t, a) \leftarrow (1 - \alpha) \cdot Q(s_t, a) + \alpha(r + \gamma \cdot \max_a Q(s_{t+1}, a)) \quad (2.1)$$

Where:

- $Q(s, a)$  is the predicted outcome for the action at a particular state  $s$
- $\alpha$  is the learning rate

- $\gamma$  is the discount factor
- $r$  is the immediate reward for achieving a state

The learning rate  $\alpha$  determines how much does the new information replace the already known information. A learning rate of 1 would mean that each training iteration the  $Q(s, a)$  is completely replaced by  $(r + \gamma \cdot \max_a Q(s_{t+1}, a))$ . A learning rate of 0 would imply that  $Q(s, a)$  is not affected by training at all. The learning rate can be adjusted during training as the model becomes more stable.

The immediate reward in the context of tic-tac-toe is 1 for a game that is won by X, 0 for a game that is drawn and  $-1$  for a game that is won by O.

# Chapter 3

## Q-Learning on tic-tac-toe

### 3.1 Generating theoretical Q-Table

The “minimax” algorithm can be used in turn-based games to calculate the best policy. It is designed to minimise the potential loss and maximise the potential gain. The algorithm works by first generating all the possible final states and then taking turns “undoing” the moves that could have lead to that state by removing **X**’s and **O**’s until the initial state (empty board) is reached. Each time an **X** or **O** is removed the worst case scenario is considered, filling up the theoretical Q-Table.

```
terminal_states = generate_all_terminal_states()

for parent_state in parent_states:

    for x/o in parent_state:

        child_state = parent_state with an x/o removed

        if child_state is terminal_state:
            Q_Table[child_state] = [game_result] * 9
        else:
            \\min or max depends on whether it's x/o removed respectively
            Q_Table[child_state][index of x/o] = min/max(Q_Table[parent_state])

        parent_states = child states
```

### 3.2 Q-Learning implementation

During the Q-Learning algorihtm batches of games were played. The agent would choose moves using a weighted probability values according to a slightly

modified boltzman function as seen in (3.1). States  $s$ , actions taken  $a$  and results  $r$  are stored in a queue. The queue is then emptied and Q-Table is updated using the Bellman equation as seen in (3.2) and (3.3).

$$p_i(\tau) = \frac{\tau^{\epsilon_i}}{\sum_{j=1}^9 \tau^{\epsilon_j}} \quad (3.1)$$

**If terminal state:**

$$Q(s_t, a) \leftarrow (1 - \alpha) \cdot Q(s_t, a) + \alpha r \quad (3.2)$$

**If non terminal state:**

$$Q(s_t, a) \leftarrow (1 - \alpha) \cdot Q(s_t, a) + \alpha \max_a Q(s_{t+1}, a) \quad (3.3)$$

As agents learn their drawrate or winrate depending on if they are playing against a perfect or random opponent increases in the form of the decaying exponential. As **X** learns against optimal **O** the draws increase in the form of a decaying exponential draws  $\% = e^{-k \cdot \text{games played}}$  as seen in Figure 3.1.

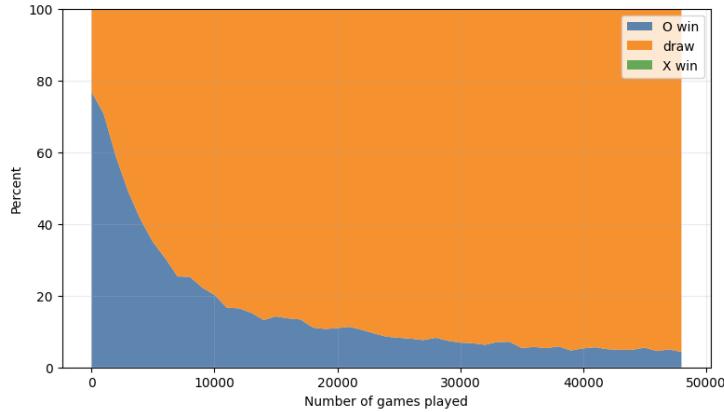


Figure 3.1: **X** training against perfect **O** causes the drawrate in the form of a decaying exponential

### 3.3 Variations in $\alpha$ and $\tau$

The values of hyperparameters  $\tau$  and  $\alpha$  matter significantly. Different models for  $\tau$  and  $\alpha$  have been tested and two models are proposed as best candidates as seen in

$$\alpha(\text{turn}, a) = \alpha_{initial} \cdot \frac{1}{1 + a \cdot \text{turn}} \quad (3.4)$$

$$\tau(\text{turn}, \text{total games}, t) = 1 + (t - 1) \cdot \left( \frac{\text{turn}}{\text{total games}} \right)^3 \quad (3.5)$$

In order to pick out the best parameters  $a$  and  $t$  for  $\alpha$  and  $\tau$  respectively, a range of  $a$  and  $t$  was picked and an agent was trained for each combination as seen in Figure 3.2 (This is actually a wrong figure, the right figure is generating overnight (cause training takes a while) but the plot would be similar).

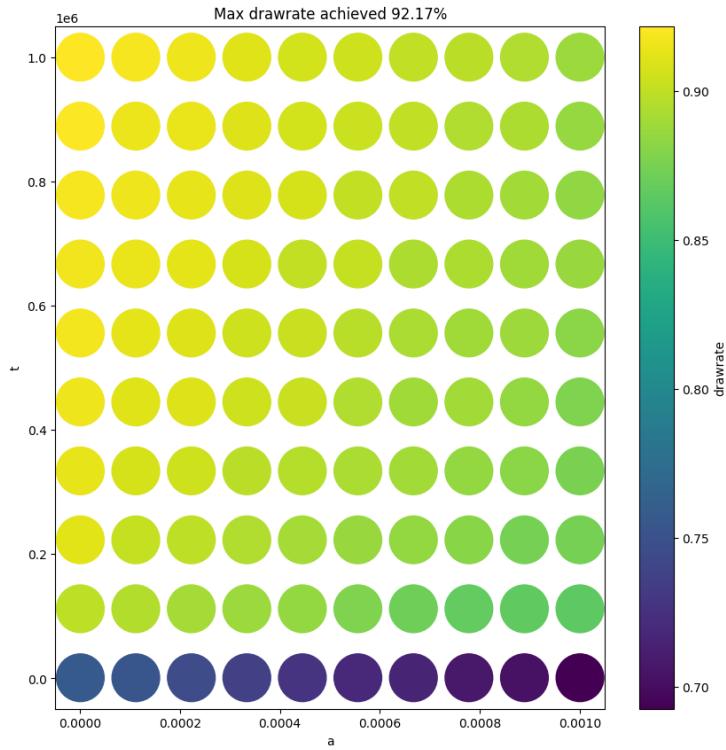


Figure 3.2: After training  $\mathbf{X}$  on 10000 batches of 10 games, the highest performing values for parameters  $a$  and  $t$  are 0.00000001 and  $10^6$  respectively

### 3.4 Optimal and non optimal opponent

### 3.5 Theoretical Q-Table compared to RL Q-Table

<b>Agent</b>	<b>X</b> trained on perfect	<b>X</b> trained on random
<b>O</b> trained on perfect	<b>X</b> winrate: drawrate: <b>O</b> winrate:	<b>O</b> winrate:
<b>O</b> trained on random	Value 2,1	Value 2,2

Table 3.1: Matrix of results of agents trained on different opponents playing against each other.

# Chapter 4

## Deep Q-Learning

**Note:** AI was slightly used for the creating of the library, specifically it helped to create the function handling the backpropogation and the function handling the loss function

**Note:** Although the library was mostly self made, in order to quickly test if it works (instead of spending a lot of time writing a game myself only to discover I have made an error with my neural network structure), I have made AI implement my library using 2 games made by Open-AI for training agents, “cart pole” and “lunar lander”

The main disadvantage of traditional Q-Learning is that it only applies to discrete states. If the input is continuous, then it will no longer work, as the Q-Table would be infinitely large. This problem is solved by Deep Q-Learning, instead of a Q-Table giving discrete q-values for actions, a neural network predicts the q-values instead.

A simple library involving numpy arrays was made in order to allow deep q-learning training.

### 4.1 Neural Network and the loss fucntion

Loss function logic goes here (4.1)

Backpropogation matrix calculus goes here (4.2)

## 4.2 cart-pole game

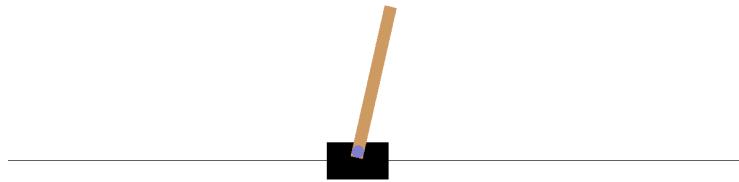


Figure 4.1: The library I made is able to train a neural network to control a cart balancing a pole. The neural network takes in 4 parameters and generates 2 outputs.

## 4.3 moon-landing game

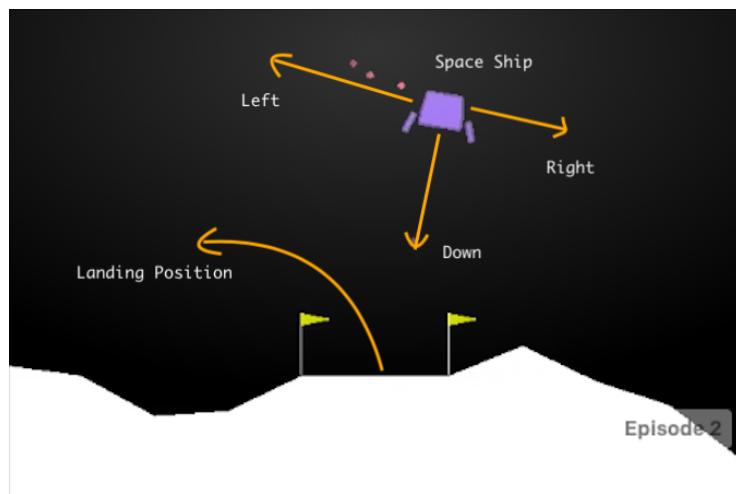


Figure 4.2: The library I made is able to train a neural network to control a space ship landing on the moon. The neural network takes in 8 parameters and generates 4 outputs.

# **Chapter 5**

# **Conclusion**

# Chapter 6

# Bibliography