

# **Q-Learning**

Peter (Petr) Ladur

56445760

Supervised by: xxx yyy

# Acknowledgements

I would like to express my gratitude to my supervisor xxx yyy for their guidance and technical feedback throughout this summer research project.

I would also like to thank the Mathematics Department at [my uni] for allowing me to undertake this summer research project.

# 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”.

# Chapter 2

## Background

### 2.1 Q-Table

Q-Table is a table which contains the total expected reward after every possible action in a particular state assuming the optimal policy is followed. 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 four already occupied positions, two actions that lead to a total expected reward of 1 (wins) and three actions which lead to a total expected reward of 0 (draws) assuming perfect play. The Q-Table can then be used to select moves which maximise the total reward, creating a perfect tic-tac-toe bot.

	(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.

$$Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha(r + \gamma \cdot \max_{a'} Q(s', a')) \quad (2.1)$$

Where:

- $Q(s, a)$  is the predicted outcome for the action at a particular state  $s$
- $s'$  and  $a'$  are the future state and action respectively
- $0 \leq \alpha \leq 1$  is the learning rate
- $0 \leq \gamma \leq 1$  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 0 if the game is not terminated. If the game is terminated, the immediate reward  $r$  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**.

The discount factor is a measure of how much the future is important compared to the current decision. For example in the context of tic-tac-toe the future matters just as much as the current situation, no matter if you lose on move 5 or move 7, you still lose. Meanwhile, in a long game with no set win condition, such as a player trying to dodge obstacles, often the most important thing is not to die right now, the future is less important.

# 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 as seen in code box 3.1. This explores the game trees and backproagates the values, filling up a theoretical Q-Table.

Listing 3.1: Minimax algorithm for generating the theoretical Q-Table

```
1 terminal_states = generate_all_terminal_states()
2
3 for parent_state in parent_states:
4
5     for x/o in parent_state:
6
7         # "undoing" a move from a state
8         child_state = parent_state with an x/o removed
9
10        if child_state is terminal_state:
11            Q_Table[child_state] = [game_result] * 9
12        else:
13            # min/max depends on whether x/o removed
14            Q_Table[child_state][index of x/o] =
15            min/max(Q_Table[parent_state])
16
17        parent_states = child states
```

## 3.2 Q-Learning implementation

During the Q-Learning algorithm batches of games were played. The agent would choose moves using a weighted probability values  $p$  according to a slightly modified Boltzmann 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).

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

Where  $\epsilon_i = Q(s, a_i)$

**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 draw rate or win rate 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 fig. 3.1. While **X** learns against a random **O** the proportion of **X** wins increases exponentially as seen in fig. 3.2.

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

The values of hyperparameters  $\tau$  and  $\alpha$  matter significantly. During the game it is best to decrease the learning rate. Initially it desired for the new information to rapidly change the Q-Table, but eventually a low learning rate means that one bad game will not significantly affect the Q-Values trained over hundreds of games in the past. Meanwhile,  $\tau$  should increase switching from exploration to exploitation. Different models for  $\tau$  and  $\alpha$  have been tested, and two models are proposed as best candidates as seen in

$$\alpha(\text{decay rate}) = \alpha_{initial} \cdot e^{-\text{decay rate} \times \text{game number}} \quad (3.4)$$

$$\tau(\text{growth rate}) = \text{Maximum Rate}^{((\frac{\text{game number}}{\text{total games}})^{\text{growth rate}})} \quad (3.5)$$

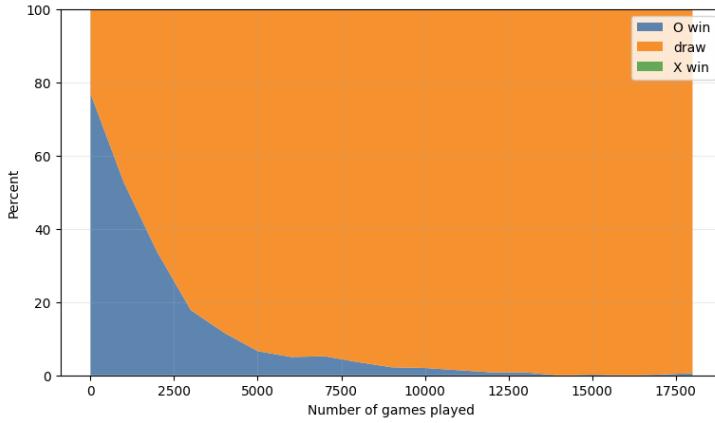


Figure 3.1: **X** training against perfect **O** causes the draw rate to increase in the form of a decaying exponential

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 fig. 3.3.

$$\alpha(\text{decay rate}) = \alpha_{initial} \cdot e^{-10^{-7} \times \text{game number}}$$

$$\tau(\text{growth rate}) = 1000 \left( \left( \frac{\text{game number}}{\text{total games}} \right)^{10^{-3}} \right)$$

### 3.4 Optimal and non-optimal opponent

After training different agents, I have made them play 10000 against each other collecting results in table 3.1.

- An agent trained against the random policy does very well against an agent trained on an optimal policy. The reason for this, is that an agent trained on the perfect policy can only choose the moves at random if its opponent makes a mistake. For example **X** trained on perfect would have never been exposed to a state where it can win, so it will not know where to go and will make a random move.
- **X** and **O** trained on a perfect opponent perform very close to a random performance in a match against each other. The likely reason for this, is that both of the opponents have been trained to draw and have not encountered the states where a non-optimal move is played.
- The likely reason for high **X** and **O** wins for agents trained on a random policy, is that they expect their opponent to continue playing the game

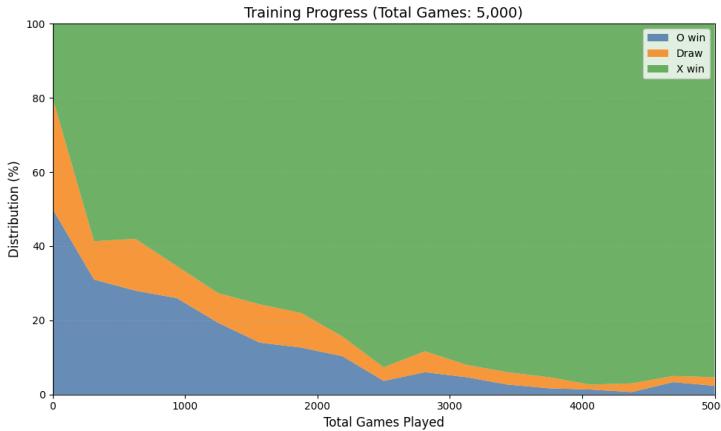


Figure 3.2: **X** training against random **O** causes the **X** win rate to increase in the form of a decaying exponential

randomly. Sometimes they might go for “risky” plays which would have a high probability of a win/draw against a random opponent, but encounter resistance against an actually trained opponent.

Agent	<b>X</b> trained on perfect	<b>X</b> trained on random
<b>O</b> trained on perfect	<b>X</b> win rate: 58.89% draw rate: 14.12% <b>O</b> win rate: 26.99%	<b>X</b> win rate: 86.05% draw rate: 8.36% <b>O</b> win rate: 5.59%
<b>O</b> trained on random	<b>X</b> win rate: 15.73% draw rate: 23.24% <b>O</b> win rate: 61.08%	<b>X</b> win rate: 43.31% draw rate: 30.02% <b>O</b> win rate: 26.67%

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

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

The theoretical Q-Table generated using the minimax algorithm has differences to the Q-Table generated through RL.

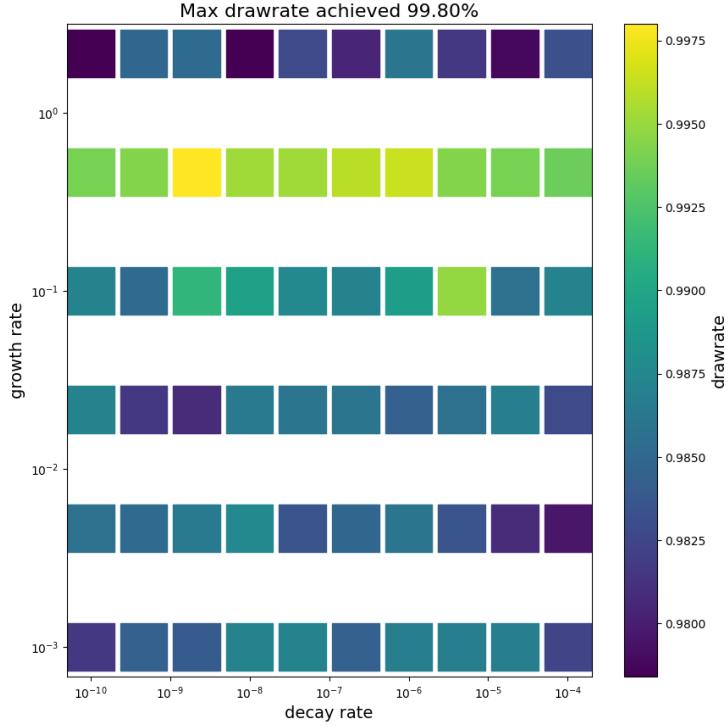


Figure 3.3: After training  $\mathbf{X}$  on 1000 batches of 10 games, the highest performing values for parameters *growth rate* are  $10^{-\frac{1}{3}}$  and *decay rate* between  $10^{-4}$  and  $10^{-10}$

When generating the Q-Table of  $\mathbf{X}$  playing against a perfect opponent,  $\mathbf{O}$  will never end up in a losing position. Hence, states where  $\mathbf{O}$  has a chance to lose are never explored and much of the RL generated Q-Table is unfilled. For the states which are explored the values of the RL generated Q-Table are similar to theoretical values as seen in table 3.2

When generating the Q-Table of  $\mathbf{X}$  playing against a random opponent, more states are explored, and two out of three actions which lead to a guaranteed win (assuming perfect play), were given the q-value of 1 as seen in table 3.3. Likely, the reason why action (2, 3) was given the value of 0.00 is because it wasn't explored enough during the exploration phase.

- - - - o - o x x	(1, 1)	(1, 2)	(1, 3)	(2, 1)	(2, 2)	(2, 3)	(3, 1)	(3, 2)	(3, 3)
Theoretical Q-Table (perfect opponent)				-1		-1	-1	-1	0
RL-estimated Q-Table				-0.75		-0.75	-0.75	-0.66	0

Table 3.2: The theoretical and RL-estimated Q-Table for x playing against perfect opponent, give similar Q-Values for explored states

- - - - - - - o x	(1, 1)	(1, 2)	(1, 3)	(2, 1)	(2, 2)	(2, 3)	(3, 1)	(3, 2)	(3, 3)
Theoretical Q-Table (perfect opponent)	0			0	1	1	0	0	1
RL-estimated Q-Table	0.062			0.00	1.00	0.00	0.06	0.02	0.83

Table 3.3: The theoretical Q-Table for x playing against a random opponent, and RL-estimated Q-Table for x playing against a random opponent give different values, because the future policy is different

## Chapter 4

# Deep Q-Learning

The main disadvantage of traditional Q-Learning is that it only works well with a few discrete states. If the input is continuous, then it will no longer work, as the Q-Table would need to be infinitely large. This problem is solved by implementing a Deep Q-Network (DQN), instead of a Q-Table. While in a Q-Table a state corresponds to all the exact Q-Values which can be taken by different actions, DQN estimates the Q-Values for all actions instead.

I have implemented a simple NumPy-based library in Python which allows for an implementation of a DQN.

### 4.1 Neural Network structure, backpropagation, cost function

By structure DQN is identical to a normal neural network. It takes in the state such as coordinates or stock prices in the input layer and forward propagates the input through the network to return the estimated Q-Values for different actions in the output layer as seen in equation (4.1).

$$\mathbf{a}^{(l+1)} = \sigma(\mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)}) \quad (4.1)$$

The main difference between a Q-Table and a Q-Network is while during training a Q-Table, the Q-Values are simply updated using the Bellman equation, in a DQN weights and biases are adjusted. With a converged Q-Table  $Q(s, a) = r + \gamma \max_{a'} Q(s', a')$  and very similarly, with a trained Deep Q-Network  $Q(s, a, \theta) = r + \gamma \max_{a'} Q(s', a', \theta)$ . During DQN training, an action picked is compared to the target as seen in equations (4.2) and (4.3). The weights and biases  $\theta$  are then adjusted through backpropagation with respect to the cost function  $\mathcal{C}(\theta)$  as seen in code box 4.1. When calculating the target, a copy of the DQN with constant parameters  $\theta^-$  is used which are updated to match the parameters  $\theta$  every N steps.

$$\text{target} = r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}^-) \quad (4.2)$$

$$\mathcal{C}(\boldsymbol{\theta}) = \mathbb{E}_{(s,a,r,s')} \left[ \frac{1}{2} (Q(s, a; \boldsymbol{\theta}) - \text{target})^2 \right] \quad (4.3)$$

Listing 4.1: Backpropagation implementation for Q-Learning

```

1   dZ[actions, batch_indices] = predictions - targets #dC/dA
2
3   #backpropagate through the layers until we get to the input layer
4   for i in range(len(weights) -1, -1, -1):
5
6       A_prev = forward_propagation_params_A[i] #previous activation
7       layer
8
9       dW = 1/m * dZ @ A_prev.T #dC/dW = dC/dA * dA/dW
10      db = 1/m * np.sum(dZ, axis=1, keepdims=True) #dC/db = dC/dA *
11          dA/db
12
13      dW_list.insert(0, dW)
14      db_list.insert(0, db)
15
16      if i > 0:
17          #this is the dA/dZ = f'(Z) * dZ
18          dZ = (weights[i].T @ dZ) *
19              functions_deriv[i - 1](forward_propagation_params_Z[i - 1])

```

## 4.2 Deep Q-Learning results

Having made a simple library for implementing a DQN, I have prompted “Gemini-3-flash-preview” to implement it in the context of a 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 q-values.

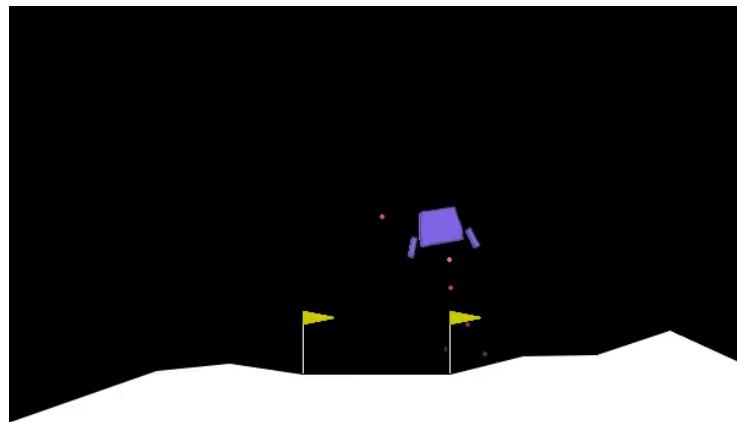


Figure 4.2: The library I made is able to train a neural network to control a spaceship landing on the moon. The neural network takes in 8 parameters and generates 4 q-values.

# Chapter 5

## Conclusion

In this project I have succeeded in implementing traditional Q-Learning in the context of tic-tac-toe and with help of AI implemented Deep Q-Learning in the context of two simple games. During Q-Learning in tic-tac-toe agents learn in an exponential manner against perfect and a random opponent drawing and winning more respectively. The most notable feature discovered when making agents trained on different policies play against each other, was that agents trained on the random policy would outperform agents trained on the perfect policy. The RL-estimated Q-Table was compared to a theoretical Q-Table was similar during training against a perfect opponent but was different against a random opponent due to differences in future policy. During the Deep Q-Learning implementation, a small NumPy-based library was developed, which could then be applied to various environments, such as the “Cart Pole” and “Lunar Lander” games developed by Open-AI.

# Chapter 6

# Future Work

During this project there were several areas which I would like to explore further.

- Experiment with other hyperparameter functions such as piecewise functions. Particularly, to not explore anything initially, and exploitation.
- Experiment with  $\gamma$ . Although it doesn't matter if you win on move 5 or move 7, winning faster is preferred, as it is still possible for an agent to make a mistake in the future
- Implementing a DQN in the context of a stock market. I would like to input historical data into a DQN and train it to sell, buy or hold, trying to make profit.

# Chapter 7

# Appendices

## 7.1 Declaration

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

## 7.2 Code

All the code for this project along with the report is accessible in a public GitHub repository. It can be accessed via this link:

<https://github.com/peterladur/Reinforcement-Learning-Research-Project>

## 7.3 References