

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 (α , τ) 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*). tion.

$$Q(s_t, a) \leftarrow (1 - \alpha) \cdot Q(s_t, a) + \alpha(r + \gamma \cdot \max_a Q(s_{t+1}, a)) \quad (1.1)$$

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 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.

	(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
- $0 \leq \alpha \leq 1$ 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 α 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.

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 the current situation, no matter if you loose on move 5 or move 7, you still loose. 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.

```
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 algorithm 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 Figures 3.1. As **X** learns against a random **O** the proportion of **X** wins increases exponentially as seen in Figure 3.2.

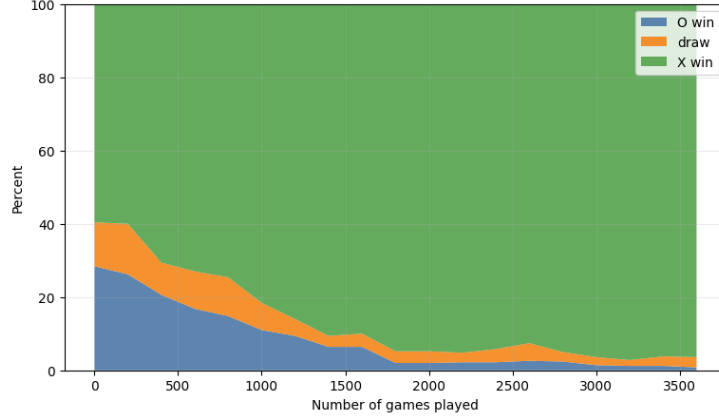


Figure 3.1: **X** training against perfect **O** causes the drawrate in the form of a decaying exponential

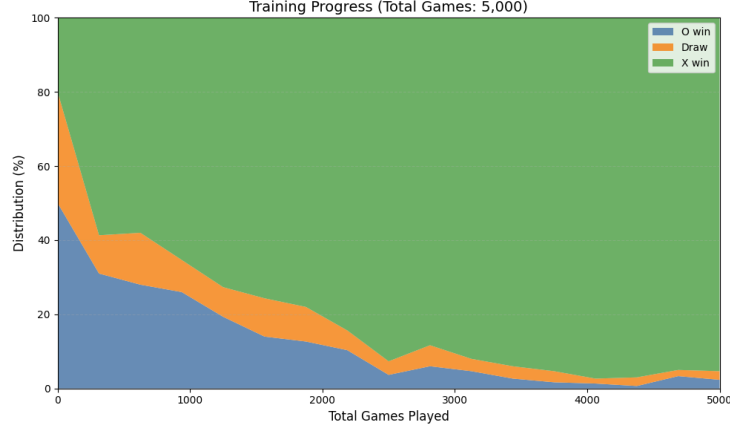


Figure 3.2: **X** training against perfect **O** causes the drawrate in the form of a decaying exponential

3.3 Variations in α and τ

The values of hyperparameters τ and α matter significantly. Different models for τ and α have been tested and two models are proposed as best candidates as seen in

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

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

In order to pick out the best parameters a and t for α and τ respectively, a range of a and t was picked and an agent was trained for each combination as seen in Figure 3.3 (This is actually a wrong figure, the right figure is generating overnight (cause training takes a while) but the plot would be similar).

$$\alpha(\text{decay rate}) = \alpha_{\text{initial}} \cdot e^{-10^{-7} \times \text{game number}} \quad (3.6)$$

$$\tau(\text{growth rate}) = 1000 \left(\left(\frac{\text{game number}}{\text{total games}} \right)^{10^{-\frac{1}{3}}} \right) \quad (3.7)$$

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

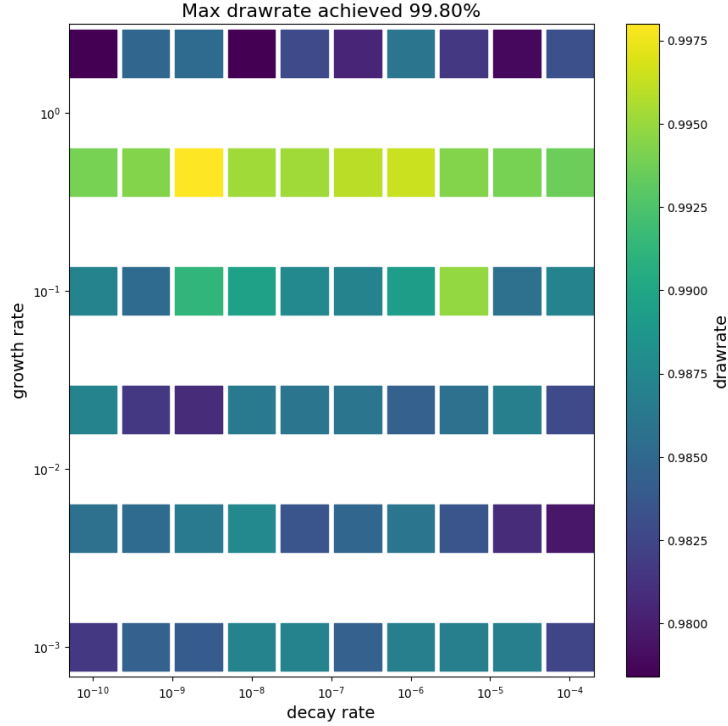


Figure 3.3: After training **X** on 1000 batches of 10 games, the highest performing values for parameters *growth rate* are $10^{-\frac{1}{3}}$ and *decay rate* between 10^{-4} and 10^{-10} respectively

trained on the perfect policy can only choose the moves at random if it's opponent makes a mistake. For example **X** trained on perfect would never have 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 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	X trained on perfect	X trained on random
O trained on perfect	X winrate: 58.89% drawrate: 14.12% O winrate: 26.99%	X winrate: 86.05% drawrate: 8.36% O winrate: 5.59%
O trained on random	X winrate: 15.73% drawrate: 23.24% O winrate: 61.08%	X winrate: 43.31% drawrate: 30.02% O winrate: 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 Q-Table

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

When generating the Q-Table of **X** playing against a perfect opponent, **O** will never end up in a losing position. Hence states where **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 **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. Although likely, the reason why action (2, 2) was given the value of 0.00 is because it wasn't explored enough during the exploration phase.

<div> <div> <div>-</div> <div>-</div> <div>-</div> </div> <div> <div>-</div> <div>o</div> <div>-</div> </div> <div> <div>o</div> <div>x</div> <div>x</div> </div> </div>	(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 Q-Table				-0.75		-0.75	-0.75	-0.66	0

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

[0. 0. 1. 0. 1. 1. 0. nan nan] [0.06251062 0.01562584 0.83249612 0.

<div> <div> <div>-</div> <div>-</div> <div>-</div> </div> <div> <div>-</div> <div>-</div> <div>-</div> </div> <div> <div>-</div> <div>o</div> <div>x</div> </div> </div>	(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 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 againsta random opponent, and RL Q-Table for x playing against a random opponent give different values, because the future policy is different

0.99995493 0. 0.06248338 0. 0.]

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 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”

The main disadvantage of traditional Q-Learning is that it only works well with a small number of 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 a Q-Table a state corresponds to all the 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, loss 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 in the output layer as seen in equation 4.1.

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

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 θ are then adjusted through backpropagation with respect to the loss function $\mathcal{C}(\theta)$ as seen in listing 4.1.

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

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

Listing 4.1: Backpropagation implementation for Q-Learning

```
dZ[actions, batch_indices] = predictions - targets #dC/dA

#backpropagate through the layers until we get to the input layer
for i in range(len(weights) - 1, -1, -1):

    A_prev = forward_propagation_params_A[i] #previous activation layer

    dW = 1/m * dZ @ A_prev.T #dC/dW = dC/dA * dA/dW
    db = 1/m * np.sum(dZ, axis=1, keepdims=True) #dC/db = dC/dA * dA/db

    dW_list.insert(0, dW)
    db_list.insert(0, db)

    if i > 0:
        #this is the dA/dZ = f'(Z) * dZ
        dZ = (weights[i].T @ dZ) *
            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.

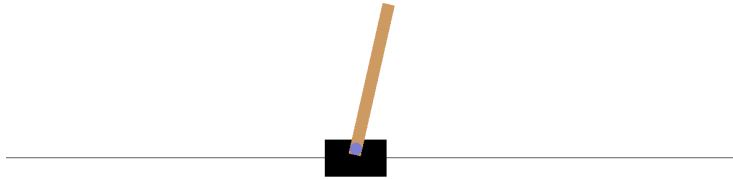


Figure 4.1: The library I made is able to train a neural network to control a cart balancing a pole. The neural network takes in 4 parameters and generates 2 q-values.

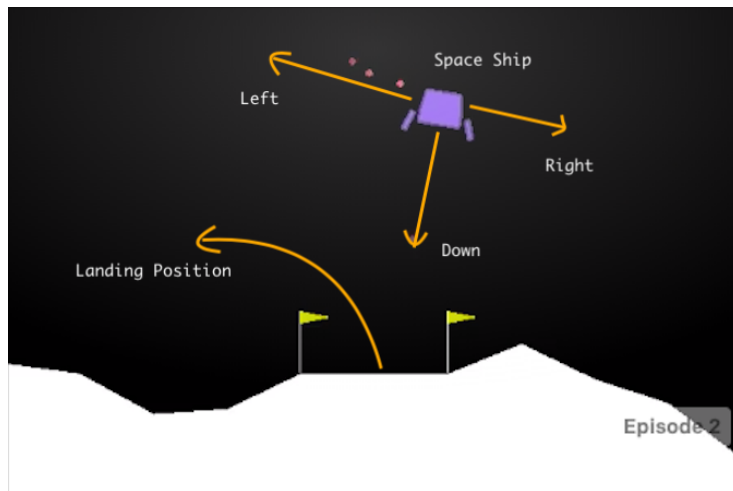


Figure 4.2: The library I made is able to train a neural network to control a space ship landing on the moon. The neural network takes in 8 parameters and generates 4 q-values.

Chapter 5

Conclusion

Chapter 6

Bibliography