Applying Reinforcement Learning to Poker
Final Project Report

Jean QUENTIN
CentraleSupélec
jean.quentin@student.ecp.fr

Elisabeth DAO
CentraleSupélec
elisabeth.dao@student-cs.fr

Timothée RIO
CentraleSupélec
timothee.rio@student.ecp.fr

Abstract

*While most games are games with perfect information and do not include that much randomness, poker is special in itself as it is a game on one hand with incomplete and imperfect information and on the other hand sometimes very dependent on luck. In practice, most players believe they lose because of the luck factor when in fact there is skill involved. The objective of our project is to make a bot to demonstrate how much people overestimate the luck factor and underestimate the importance of learning a proper strategy by building a poker playing robot with reinforcement learning. All implementations are available in our GitHub repo: **timrio/reinforcement-learning-poker**.*

1. Introduction

1.1. Motivations

Games have always been a field of interest for Reinforcement Learning as they often follow simple rules but require some degree of complex strategies to play them at a competent level.

Poker as part of the entertainment industry has known growing popularity over the years, be it in casinos themselves or now directly on the web were it became so popular that it even received high media coverage as a conventional sport.

The first motivation of this project was first the challenge it represents. Some good advances were made, specially in games like Chess or Go game which are deterministic (no randomness involved) games with complete information (players have access to all the information they require). The research has then shifted into stochastic games with incomplete information, like Poker. It is stochastic because the shuffling of cards introduces randomness into the game and it is a game of imperfect information, as players cannot see their opponents' cards, forcing players to make decisions based on hidden information. Given the relatively simple rules of the game, there is an enormous amount of subtle and sophisticated scenarios that can occur during a hand of play. Moreover, Poker is an inherently psychological game. It is crucial to have an understanding of your opponents and how they think to be able to play well. Therefore, robots should integrate a probabilistic reasoning and opponent modelling to become competent, which make Poker a challenging domain for Reinforcement Learning. What's more, the challenges brought by non-deterministic games with hidden information are much closer to those present in real world applications and therefore, a solution that can be applied to a game with those characteristics is one that can possibly be adapted to solve real world problems.

Another key goal of this project is to show that skill is an important factor to win in Poker. If a robot is capable to increase its chances to win through the games.

The focus of our work is to create agents than can learn how to play poker from scratch. The Poker variant Texas Hold'em was used as our test base.

1.2. Related work

Remarkable results were achieved in games research such as the well-known Deep Blue Computer, which was the first computer to ever defeat a human chess champion. However, such success has not yet been achieved for incomplete information games. This is because the game state is not fully visible which means that there are hidden variables / features. Therefore, decision making in these games is more difficult because predictions about the missing data must be made. This makes it almost impossible to obtain an optimal solution.

Research on the use of reinforcement learning for Poker has been active for over 20 years. Several approaches were made and in this section, we will present a brief overview of the majors notable advances in building a poker playing computer agent.

A first computer-based approach was based on quality of the hand calculation based on the strength of the card combination together with the probability of a better hand resulting of the current hand. Other area developed opponent modeling where the objective is to identify the type of opponent and adapt the agent's playing style to improve its performance against those opponents. The great breakthrough in Poker research began with the use of Nash's equilibrium theory. Other recent methodologies were based on on the Monte Carlo Search Tree algorithm [9]. A successful work closely related to this approach is [8]/ It presents a reinforcement learning methodology to another simplified version of Poker. This approach uses Q-Learning to learn how to play against several opponent types. Last explored methods used deep reinforcement learning [3]. Despite all the breakthroughs achieved, there is no known approach in which the agent has reached a level similar to a competent human player

2. Problem definition and Modelisation

2.1. Poker Rules

Poker Texas Hold'em (No-Limit) is one of the most popular forms of poker. It is also a very complex game because there are several factors that make the game as uncertain environment (concealed cards, bluffing). Each players starts with a given amount of chips and two cards, the first player at the left of the player that dealt the cards is the small blind, and the second on the left is the big blind, as shown in Figure 1. They are called this way since both players are forced to bet (half the minimum betting amount for the small blind and the full minimum for the big blind) before even seeing their private cards.

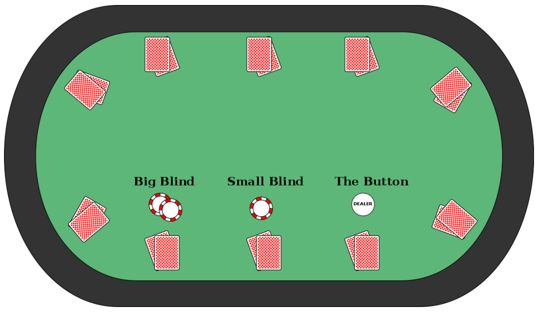


Figure 1: Player position (taken from [10])

Once the blinds have played, the first two cards are dealt and the first player at the left of the big blind will start betting. Players are asked to bet on four occasions that correspond to cards being dealt:

- Pre-flop: the players have been dealt their two private cards and are awaiting the next 3 (called the "Flop").
- Flop: the players have seen their two private cards as well as the first 3 that are public (Flop) and are awaiting the fourth card (the "Turn").
- Turn: the players have seen 4 public cards and are awaiting the fifth called the "River"
- River: the players have seen all public and private cards and can bet one last time before the "showdown" when all players that haven't folded show their cards.

At each of the mentioned steps, players are asked to play, which consists in one of the following actions:

- "Betting any amount" (or "Raising") above what has already been bet (if you bet all you have you are "all-in")
- "Following" (or "Calling"), which means betting exactly as much as the maximum that has been bet on this round
- "Folding", which means not betting and withdrawing from the current round

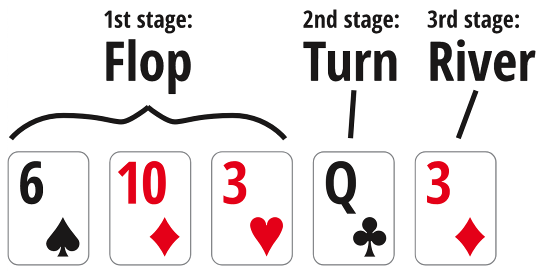


Figure 2: Card distribution (taken from [2])

If there remains more than one player at the showdown (more than one players haven't folded), the remaining players show their two private cards and the dealer shows the strongest 5-card combination using any of the 5 public and 2 private cards. As you can see in figure 9 of Appendix A, the possible card combinations are (starting from the highest rank) Straight flush, Royal flush, Four of a kind, Full-house, Flush, Straight, Three of a kind, Two pair, One pair, High card.

Poker is then a multi-round game, where each player can choose to bet or fold at every round, and the game continues until either all players but one go out of chips or until the players decide to stop. The ”winner”, if there is a need to define one, is then the player with the most chips in the end.

2.2. Modelisation

Having defined the rules of poker we can clearly understand that trying to solve this problem for any number of players and the complete set of actions with regular reinforcement technique would require an enormous amount of RAM to store the Q-table and nearly intractable computations to update it.
We have thus decided to simplify this problem by taking the enforcing the following restrictions :

- We only consider two players games (this is called a ”heads-up” game in poker)
- Players can only perform two actions: folding and going all-in (which in practice means that players only see their private cards, and must either go all-in or fold in pre-flop).

3. Environment

Since the poker game is relatively complicated, we needed to get access to a pre-built environment that would allow us to run as many episodes as needed in a computationally efficient manner.
To solve this issue, we have resorted to **clubs**[5] is a python library for running arbitrary configurations of all variants of the poker game, including the Texas Hold'em No limit that we focus on during this project. It enables us to run visuals as shown in figure 3. Although the library already follows the OpenAI gym interface, we decided to use the gym package wrapper **clubs_gym**[6] that enables us to use the familiar syntax of gym.

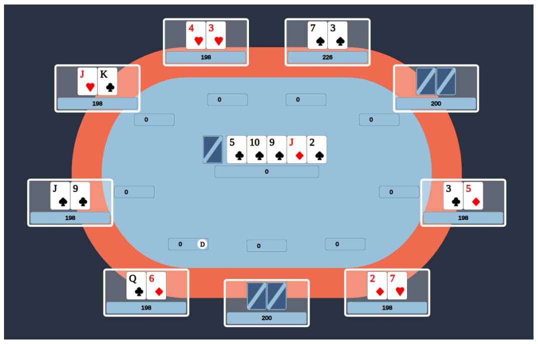


Figure 3: A screenshot taken from the clubs python library [5]

As a side note, we had the pleasure to discuss with **clubs** and **clubs_gym** creator Ferdinand Schlatt who kindly fixed the GitHub issue [4] opened on the repository of **clubs_gym** following a bug we found in his implementation.

4. Agents

At first, we have tried to implement a model that play in a simpler environment with only two players and two possible actions for each player: push or fold. We have started by implementing and comparing[1] standard reinforcement learning approaches. [7] [1]. Then, we have tried out a Deep Reinforcement Learning approach to try to solve our environment size issues. We have thus implemented three agents:

- Q-learning Agent for single-round poker (Section 4.2)
- Q-learning Agent for multiple-round poker (Section 4.3)
- Deep Q-learning Agent for single-round poker (Section 4.4)

4.1. Environment used in our approach

4.1.1 State Space

We have chosen two ways to describe the state space: either with only (card1, card2) or with (card1, card2, stack_player_1, stack_player_2). It is clear then that the second environment parametrization takes into account the distribution of chips among the three possibilities (chips either belong to player 1, player 2 or are on the pot but we only need to know two of those since poker is a zero sum game [2]).
We will discuss the different implications of these environment parametrizations in the next parts of the report.

4.1.2 Action Space

The action space is relatively simple since there are only two possible actions: either fold or go all-in.

4.1.3 Rewards

The rewards are quite natural in this game: it is the amount of chips you gain or you lose at each round.

[1] time required (in seconds and number of iterations) to train the agents up to a significant win rate against a naive bot, and make the algorithms fight against each other at the end

[2] Poker is a zero-sum game by design but in online poker the house takes a cut of the pot at each round thus this should be taken into account if the goal is to make an online poker-playing bot

$$Q[s_t, a_t] := (1-\alpha) Q[s_t, a_t] \\ +\alpha \left(R_{t+1} + \gamma \max_{a_{t+1}} Q[s_{t+1}, a_{t+1}] \right) \tag{1}$$

Figure 4: Update formula for the Q-table of a Q-learning agent

4.2. Q-learning Agent for single-round

In this section we will focus on creating a Q-learning agent that learns to play single-round poker. As mentioned in Section 2.1, at each round, the two players at the left of the dealer are the small and big blind respectively. In the heads-up configuration we play in, that means that at each round you are either small blind or big blind. This has an impact on the behavior of our agent since a BB agent plays in second and knows what if the other player has fold or gone all-in, and thus has more information.

4.2.1 Environment

We will use the (card1, card2) parametrization of the environment in this part, since it makes the environment size relatively small and the strategy is easier to learn. However we needed a way to tell our agent that if is big bling or small blind since this impacts its behavior: we could have increased the dimension of the state space but we decided to train two agents to make it simpler: a SB agent and a BB agent.

4.2.2 Agent

The first agent we have implemented is thus a Q-learning agent based on learning a Q-table that learns a table containing the expected reward of the agent in each state. Starting from zero, the values of this table are updated using the following Equation 1.

4.2.3 Results and discussion

Since the state space dimension is relatively small, we were able to run the training of the two agents quite quickly. We experimented with three types of step sizes (either 1/N, 0.01 and 0.1) as shown in Tables 5 and 6.
As we can see, the agents corresponding to step sizes 0.01 and 1/N for both BB and SB (5a and 5b for SB, 6a and 6b for BB) perform quite well and learn a wining strategy against our random policy agent. The main difference between the two strategies is that with a fixed step-size of 0.01, the agent pays more interest in the recent experiments and is thus able to adapt to a changing environment. However, in our case the environment doesn't change and the agent is more sensitive to the random noise that can appear since poker is a random-based game, this explains why it is more likely to make poor decisions and loose chips as shown in generations 70-75 of Figure 5b. On the other hand, the 1/N step size agent pays as much as attention to all previous steps and is thus more efficient in this case, where the environment doesn't change: it learns a policy that is more likely to win both as a big and a small blind (5a and 6a respectively).

Finally, the 0.1 step size agents (5c and 6c) often get stuck in local minima: they start by learning a bad policy by lack of luck in the first rounds and since the step size is too big it explores too little and keeps folding with good hands. It thus learns a policy that is so bad that playing randomly would be more effective.

To conclude, single round poker is the typical example of a reinforcement learning problem where setting the step-size to 1/N is the most efficient strategy, since the environment doesn't change during the course of an experiment.

4.3. Q-learning on multiple rounds

Though the last agents obtained very promising results on single-round poker, this game is still far off from the actual Texas-Hold'em game. In order to define a game that resembles more real poker, we decided to implement multi-round poker games. To do that we then need to integrate the notion of stack size (how much money each player has at each round), which will then let the agent know if he is BB or SB by calculating the chip differential at each round. Another key point is that stack size is a major part of the agent's decision to fold or not: if your stack is close to 0, folding is not free since you lose the big/small blind that can start making up a big fraction of your total stack. However is you have your whole stack, losing the blind isn't much of an issue.

4.3.1 Environment

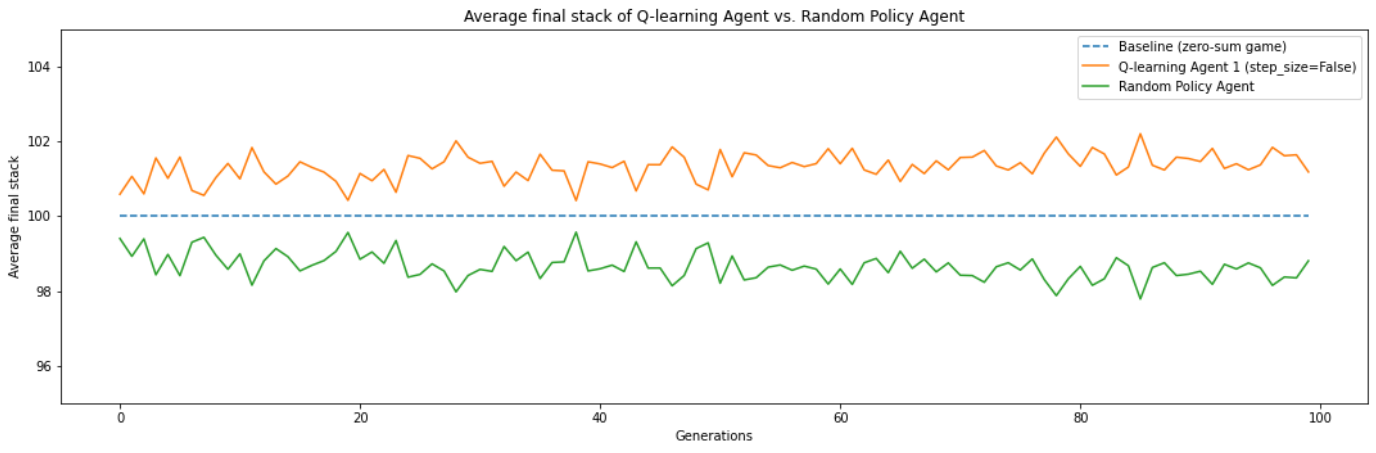
The state space of this agent is thus described by the vector (card1, card2, number of chips) and can get very big quite quickly as we will see in the next parts.

4.3.2 Agent

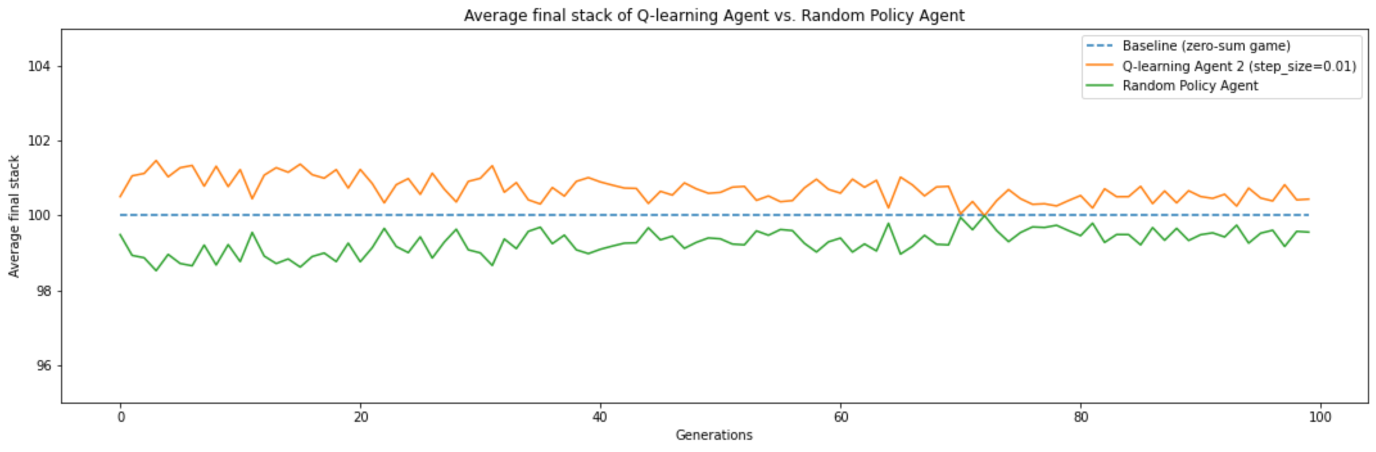
The agent is basically the same as in the previous section, the state-space is the only thing that has changed.

4.3.3 Results and discussion

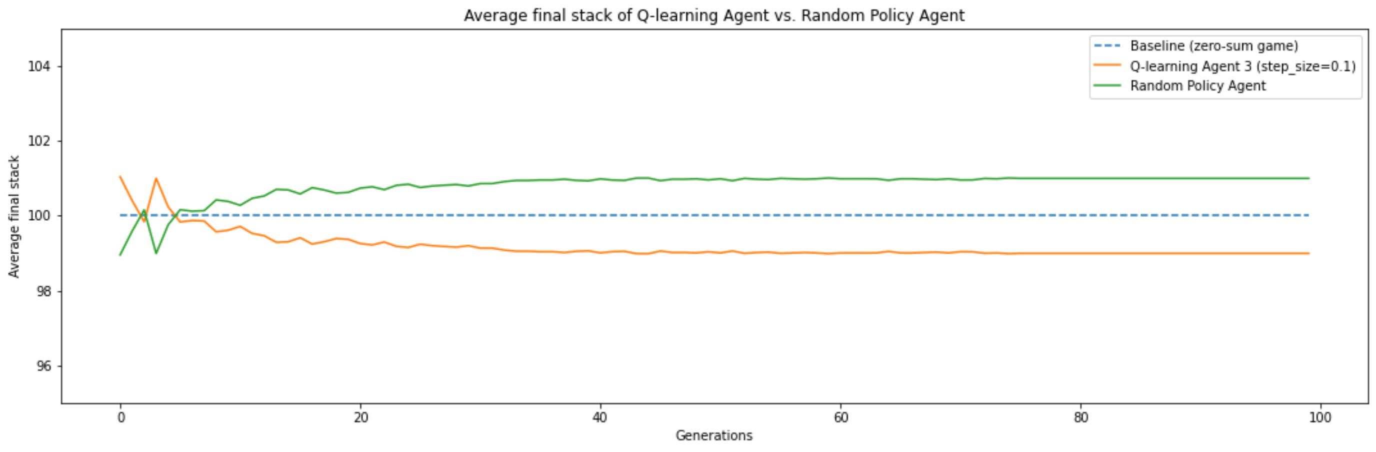
As shown in Figure 7, we obtained quite poor results, which is due the extreme size if the environment that we defined in the previous section (4.3.1). Adding the stack sizes to the dimension of the state space makes it nearly intractable for a consumer grade laptop, even with multiple days computing



(a) step-size = 1/N

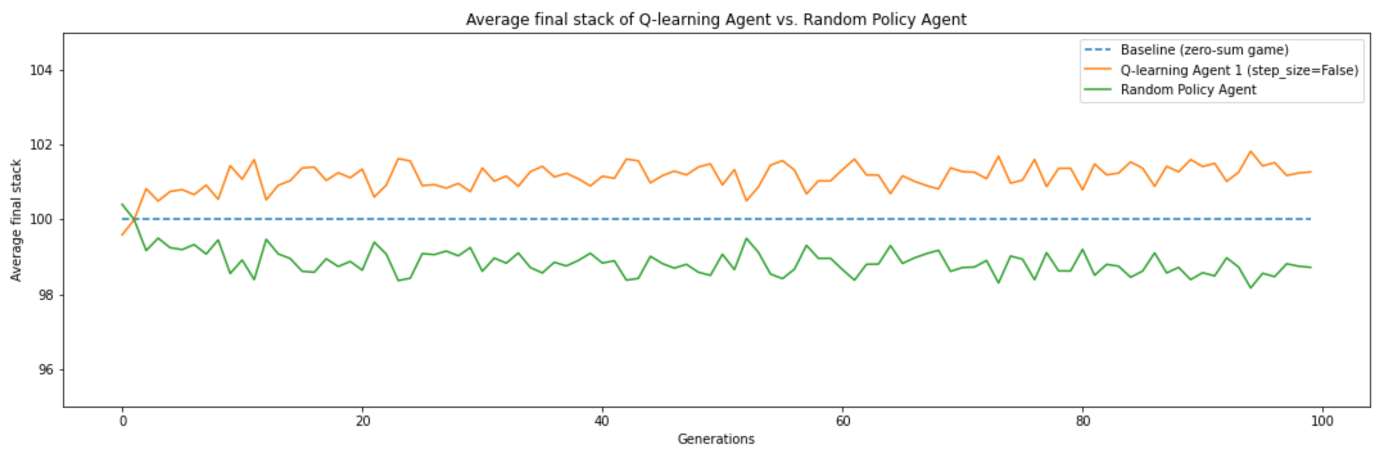


(b) step-size = 0.01

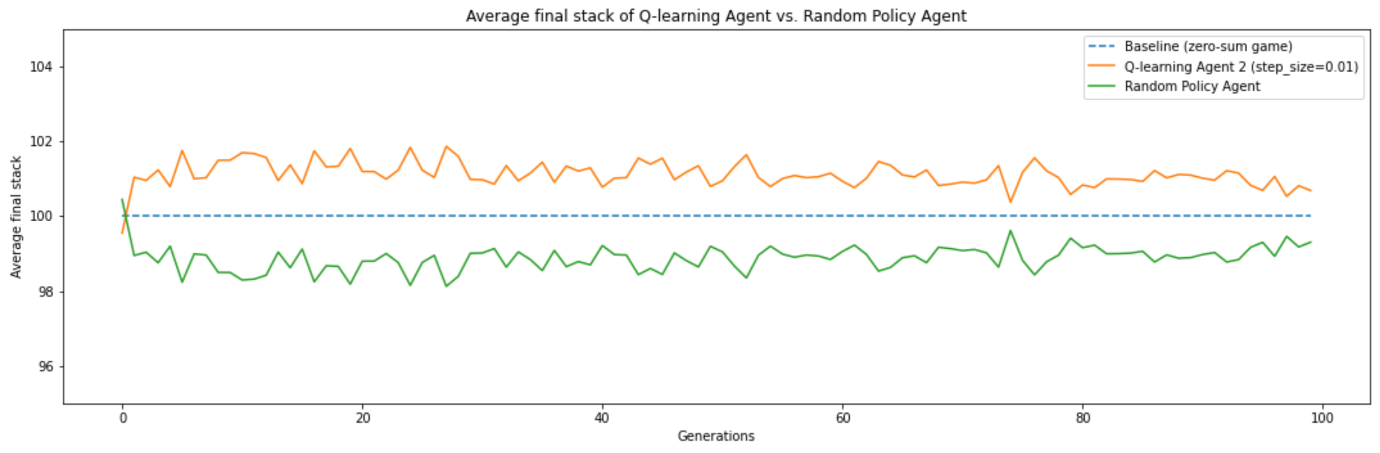


(c) step-size = 0.1

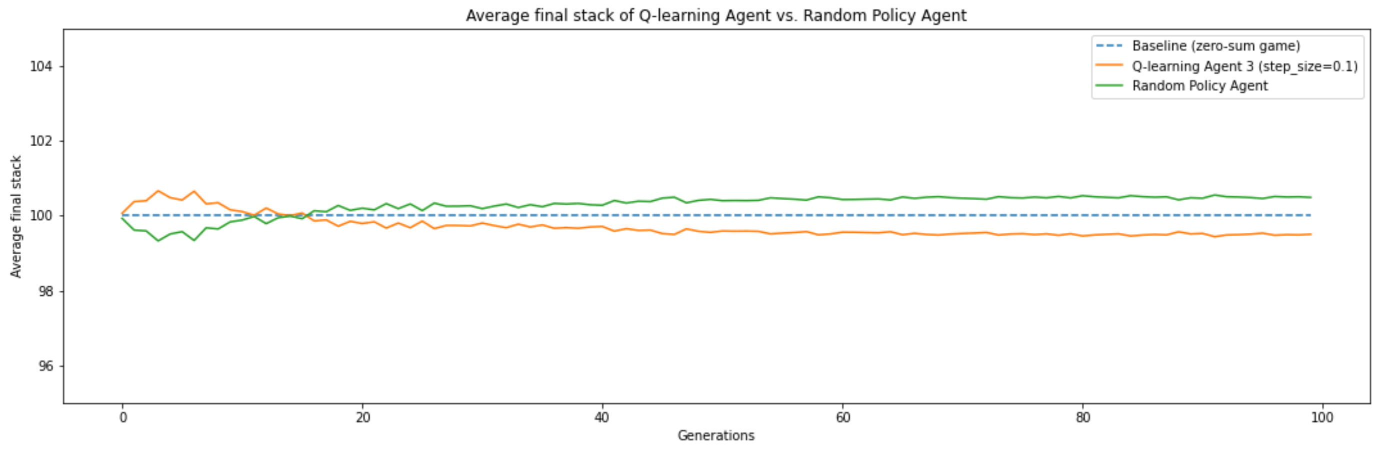
Figure 5: Average reward for a Small Blind (SB) Q-learning agent (orange) with different step-sizes vs a random policy agent (green) for single-round poker games by number of episodes played



(a) step-size = 1/N



(b) step-size = 0.01



(c) step-size = 0.1

Figure 6: Average reward for a Big Blind (BB) Q-learning agent (orange) with different step-sizes vs a random policy agent (green) for single-round poker games by number of episodes played

times. A solution we imagined would be to discretize the stack sizes even more: knowing you have 40 chips and the opponent 60 or 45/55 is basically the same thing thus we could imagine defining bins of stack splits (you have less than 50 chips/the opponent has more than 50, you both have less than 50, etc.) to reduce the state space dimension.

4.4. Deep Reinforcement Learning

4.4.1 Basic Principle

Next, we decided to implement deep reinforcement learning, that leverages the learning capacity of deep neural networks to tackle complex problems.

With deep reinforcement learning, we represent the various components of agents, such as policies $\pi(s, a)$ or values $q(s, a)$, with multi-layer neural networks. Instead of having a Q-matrix that stores every possible state and actions we use a Q-network that takes the state of the environment as input and outputs the predicted q-value of each action. The parameters of these networks are trained by gradient descent to minimize loss function. The deep network approximate the actions values for a given state S_t.

4.4.2 Encoding

Of course for this problem we have used one hot encoding, that is the process of converting the categorical data variables to provide them to our network. Each card is represented by a value. Our one hot encode card function is built as follows : every symbol is represented by a number (heart is represented by 0, club is 1, diamond is 2 and spade is 3) and each rank is represented by a number.

4.4.3 Replay Memory Buffer

At each step, the agent selects an action ϵ - greedily with respect to the actions values and adds a transition to a replay memory buffer. The replay buffer stores trajectories of experience when executing a policy in the environment. We store the agent's experiences at each time step in a data set called the replay memory with finite size limit. This replay memory data set is what we randomly sample from to train the network. We use this technique of replay memory in order to break the correlation between consecutive samples because otherwise, if the network learns only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and it would therefore lead to inefficient learning. Taking random samples from replay memory breaks this correlation.

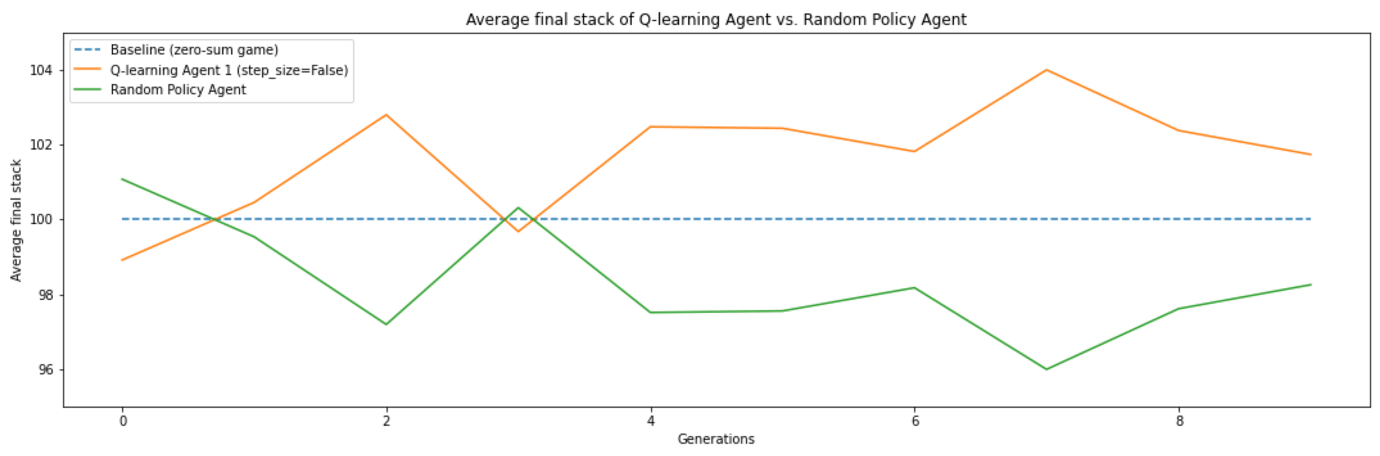
4.4.4 Target Network

Basically, for a single sample, the first pass to the network occurs for the state from the experience tuple that was sampled, to calculate the Q-value for the relevant action. The network outputs the Q-values associated with each possible action that can be taken from that state, and then the loss is calculated between the Q-value for the action from the experience tuple and the target Q-value for this action. Then we do a second pass in order to calculate the target Q-value for this same action. The aim is to get the Q-value to approximate the target Q-value. As we should not use the same network to calculate both of these values, we introduce another completely separate network to obtain the target Q-values, called the target network. It is basically a clone of the policy network. Its weights are frozen with the original policy network's weights and we update the weights in the target network to the policy network's new weights every certain amount of time steps (100). The target network will in turn update the target Q-values with respect to what it has learned over those past time steps. This target network provides us the max Q-value for the next step, plug this value into the Bellman equation in order to calculate the target Q-value for the first state.

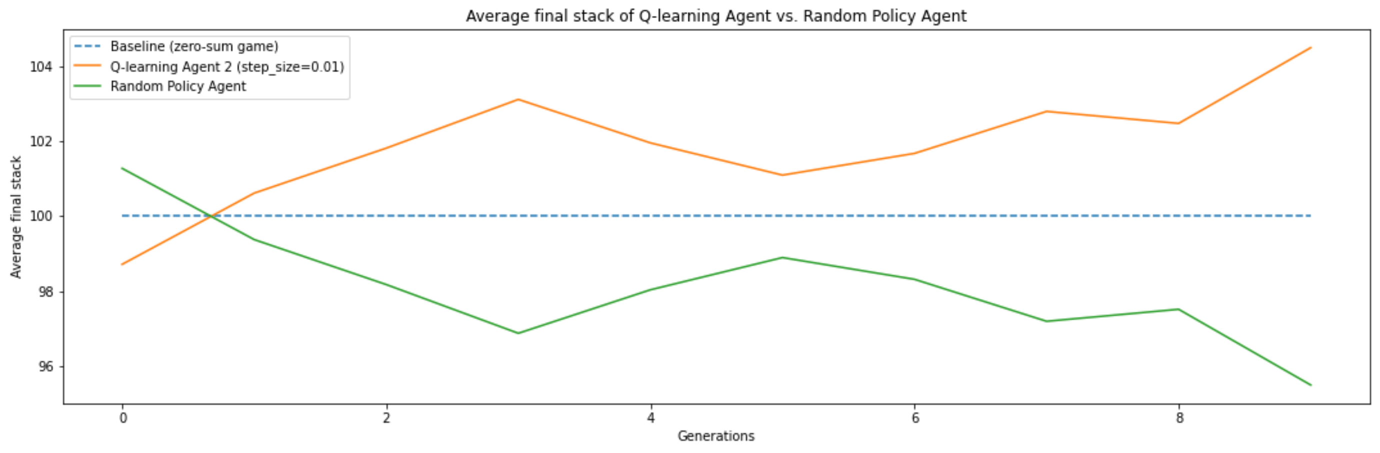
4.4.5 Methodology Wrap-up

Here is a basic wrap up of our approach :

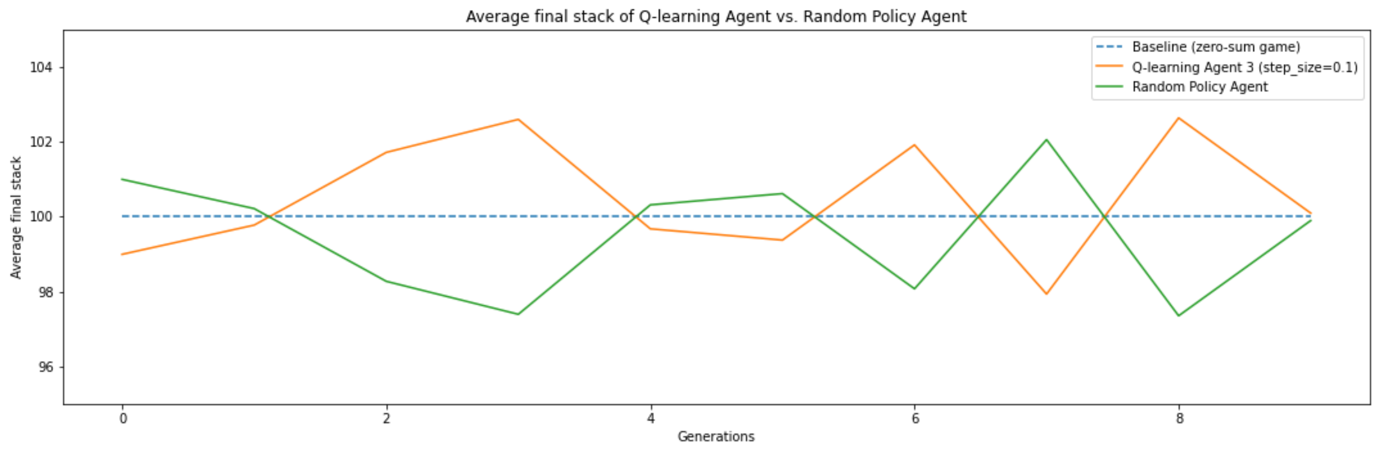
1. Initialize replay memory capacity
2. Initialize the network with random weight
3. Clone the policy network and call it the target network
4. For each episode :
    - Initialize the starting state
    - For each time step
        - Select an action (exploration and exploitation)
        - Execute selected action
        - Observe reward and next state
        - Store experience in replay memory
        - Sample random batch from replay memory
        - Preprocess states from batch
        - Pass batch of preprocessed states to policy network
        - Calculate loss between output Q-values and target Q-values (it requires a second pass into the network to calculate the max Q-value for the next state across all possible next actions)



(a) step-size = 1/N



(b) step-size = 0.01



(c) step-size = 0.1

Figure 7: Average reward for a Q-learning agent (orange) with different step-sizes vs a random policy agent (green) for multiple-round poker games by number of episodes played

- Gradient descent updates weights in the policy network to minimize loss (weights in the target network are updated to the weights in the policy network)

However, we obtained rather unsatisfying results as you can see below :

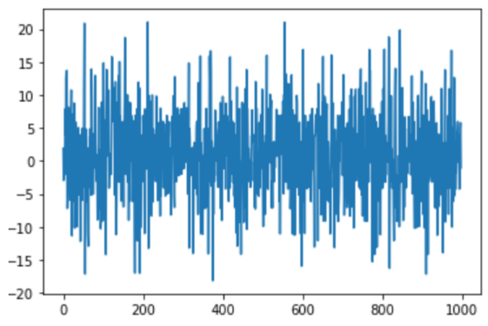


Figure 8: DQN Agent rewards over the training

A. Poker Rules

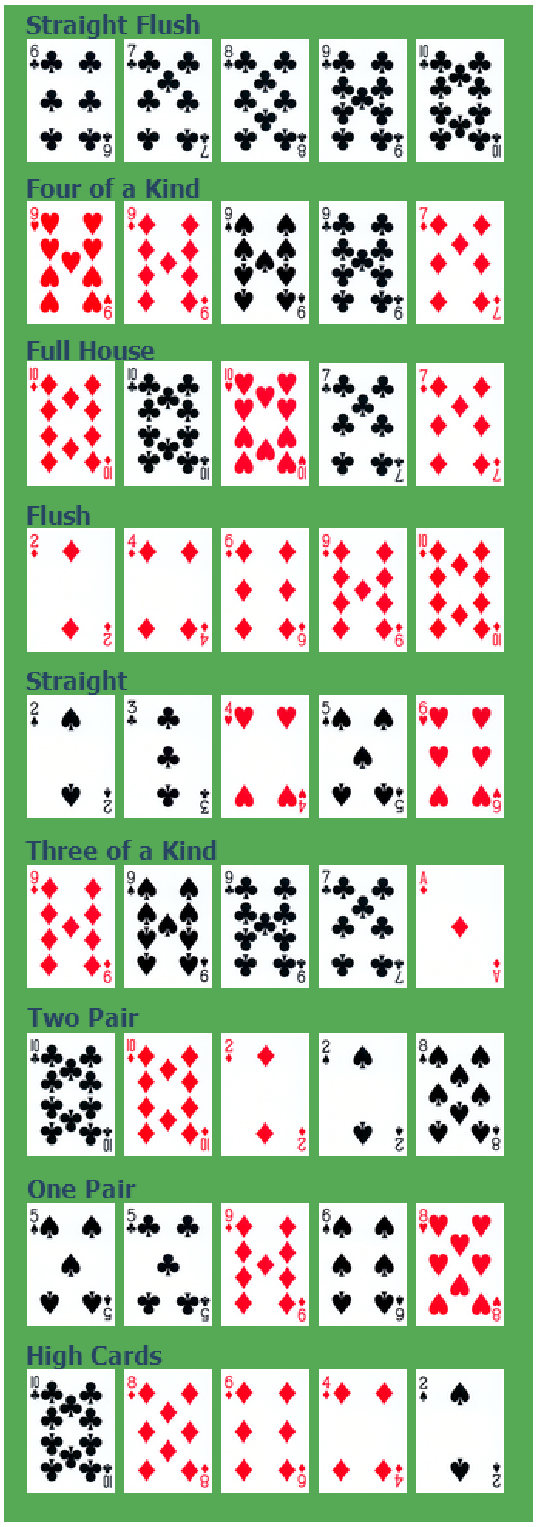


Figure 9: Ranking of hands in poker (taken from [10])

References

[1] Fredrik A. Dahl. A reinforcement learning algorithm applied to simplified two-player texas hold'em poker. In Luc De Raedt and Peter Flach, editors, *Machine Learning: ECML 2001*, pages 85–96, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 3

[2] Anita Guo. LES RÈGLES DU TEXAS HOLD'EM POKER. https://medium.com/@anitaguo49/simple-strategy-for-texas-hold-em-poker-starting-hands-d7bfa7de9502, 2020. [Online; accessed 25-March-2022]. 2

[3] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. 03 2016. 2

[4] Jean Quentin. Only one hole_card is displayed in env.reset() (nolimitholdemtwoplayer-v0). https://github.com/fschlatt/clubs_gym/issues/4?fbclid=IwAR0gqXMV6GR6RxWapSxeU6jueqEZ6os_7ic5JBH4hl-W6HfRqqg_7Lznc3g, 2022. 3

[5] Ferdinand Schlatt. clubs. https://github.com/fschlatt/clubs, 2022. 3

[6] Ferdinand Schlatt. clubs_gym. https://github.com/fschlatt/clubs_gym, 2022. 3

[7] Csaba Szepesvári. *Algorithms for Reinforcement Learning*, volume 4. 01 2010. 3

[8] Luís Teófilo, Nuno Passos, Luís Reis, and Henrique Lopes Cardoso. Adapting strategies to opponent models in incomplete information games: A reinforcement learning approach for poker. volume 7326, pages 220–227, 06 2012. 2

[9] Luís Filipe Teófilo and Luis Paulo Reis. Identifying playerś strategies in no limit texas holdém poker through the analysis of individual moves, 2013. 2

[10] Wikipedia. Texas Hold'em Rules. https://fr.wikipedia.org/wiki/Texas_hold%27em, 2022. [Online; accessed 25-March-2022]. 2, 10