Instituto Superior Técnico

Mestrado Integrado em Engenharia Mecânica

Intelligent Systems

Project 12

Using Reinforcement Learning to play Blackjack

Group 6

André Fabião 87148

Diogo Oliveira 87176

José Trigueiro, 87225

Faculty: Susana Vieira, João Sousa, Miguel Martins

2020 – 2021

14th of December 2020

Contents

1. Introduction

In the Intelligent Systems course, the focus was on two main types of Computational Intelligence paradigms: *supervised* and *unsupervised learning. Supervised learning* consists in training from a dataset with labels (the correct action to take by the algorithm) provided by an external source, while *unsupervised learning* deals with the discovery of hidden knowledge and patterns in data. *Reinforcement learning* provides a third approach to the problem of machine learning, where the algorithm learns not through an external supervisor or by hidden features in the data, but instead by the reward it gets from interacting with an external environment.

In this work multiple reinforcement learning algorithms were implemented and tested, using a simplified version of the card game of blackjack as an environment. In a first approach, two more classical methods were tested: Monte Carlo methods and Q-Learning. Next, a Deep Q-Learning approach using Neural Networks was also implemented.

2. Reinforcement Learning Fundamentals

As described in [1], all types of reinforcement learning algorithms involve an *agent*, which acts as both a learner and a decision maker, and the *environment*, which represents everything around the agent that it can sense and interact with. The agent receives information about the *state* of the environment and takes an *action,* accordingly, changing the state of the environment. While the agent faces uncertainty about the environment, it seeks to reach a certain *goal* by its actions, which is to maximise over time the *reward* that it receives from the environment. While maximizing rewards at each instant models an instant benefit, a *value function* may also be considered, modelling the long-term benefit of a certain action, meaning the value of a state will be the expected reward the agent will receive in the future following that state.

The mapping between environment states and the action that the agent should take when faced with those states is called a *policy*. This means the problem will be reduced to finding the policy that maximizes the value function over the possible environment states.

The most general model for a Reinforcement Learning system is shown in Figure 1, based on the mathematical definition of a Markov Decision Process. Given a certain state S_t, the agent selects an action A_t. The action causes the environment to yield a reward R_{t+1} to the agent, and to change its state to a new one, S_{t+1}

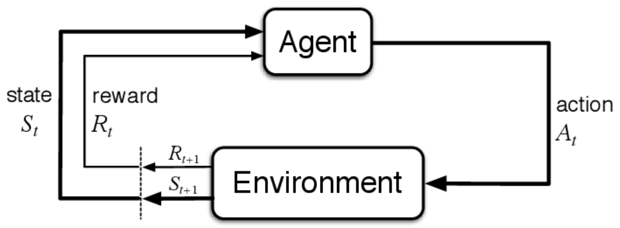


Figure 1 – Reinforcement Learning system

To find non-approximate solutions to the problem, the *Markov property* should be assumed: the state of the environment must include all information about past interactions between agent and environment relevant to the future. This is valid for both Q-Learning and Monte Carlo methods. Using this notation, the *return* of a sequence of events may also be defined:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1}$$

In the above equation γ is the *discount rate*, penalizing rewards in later terms in the sequence, and T is the final time step in the algorithm. It is then finally possible to define, as in [1], the value function associated to a policy π and a state s, using all future rewards r, states s' and actions a:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')], \;\; for\ all\ s \in S \tag{2}$$

The *optimal* policy, which may not be unique, is then the policy that maximizes $v_\pi(s)$, $v^*(s)$ for all states $s \in S$. Associated with it is the *optimal state-action value function,* $q^*(s,a)$. Both functions are defined by Bellman optimality equations:

$$v^*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v^*(s')] \tag{3}$$

$$q^*(s,a) = \max_\pi q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q^*(s',a')\right] \tag{4}$$

An agent may be able to learn an optimal policy, but this may require a long learning time and high computational cost, so reinforcement learning more commonly deals with approximations that tend to an optimal policy instead.

One of the key issues that reinforcement learning deals with when approximating an optimal policy is *maintaining exploration*, meaning the agent should be encouraged to follow policies that it tried before and got a high reward, while keeping the ability to steer of those policies and try new ones to find possible better actions. This is the trade-off between *exploitation* of known experience to obtain reward, and *exploration* of new actions to get better experience.

3. Blackjack environment: description and simplifications

One of the most important steps of creating a reinforcement learning solution is defining an appropriate simulation environment to train the agent. In this case, the Blackjack card game was used, for its small state-action space and direct formulation as a Markov Decision Process.

In Blackjack, the dealer and player are dealt two cards, with one of the dealer's cards facing up and the other facing down, and both cards of the player facing up. Regular playing cards are used, with cards 2 to 10 having their numeric point value, the denominated "face" cards (jack, queen, and king) being worth 10 points each and the ace counting as either 1 or 11, depending on what is most beneficial to the player. If the sum of the player's cards is 21, they are said to have a "natural", meaning they instantly win unless the dealer also has a natural, in which case the game is tied. If the player does not have a natural, they may choose to "hit," requesting an additional card or to "stick" where they stop playing and allow the dealer to draw cards. The player may hit as many times as they want, but they instantly lose the game if they "go bust," by having in their hand a value higher than 21. If the player sticks, the dealer draws cards until they reach the sum of 17 of higher. If the dealer goes bust the player wins, otherwise the winner is the person who has a value closer to 21 in their hand. If the dealer and the player have the same score in their hand, the game ends in a draw

The environment chosen is a default environment from Python reinforcement learning toolkit OpenAI Gym [2]. This environment simplifies the standard casino blackjack game in its implementation:

- No advanced player actions (double down, where the bet is doubled with the cost of only being able to hit once and split, where the player can split their hand in two if they have two equal cards as if they were two different players) are allowed, only hitting, or sticking.
- An infinite deck is assumed, meaning the probability of drawing a certain card is the same regardless of the previously drawn cards.
- The bet is assumed to be constant throughout all games, with the value lost in a losing game being the same as the gains in a winning game. The only exception is when drawing a natural, in which case the reward is 50% higher than in a regular win.

The game of blackjack may then be defined as a finite, episodic Markov Decision Process, where every game is a different episode that ends when the game is finished. The reward is +1.5, +1, 0, or -1, if the player wins by drawing a natural, wins in any other way, draws, or loses. No discount is applied ($\gamma = 1$) and no reward is attributed mid-game, so the terminal reward of the game is the return. The state depends on two factors: the dealer's face up card and the player’s cards. It is obvious from the rules that the player should always hit when the sum of his cards is 11 or lower, since he may never go bust in this situation, meaning its unnecessary to include those states in the model. The choice also depends on if the player has an ace that may count as 11 without going bust (called *usable*). The possible states of the environment are then defined as a three-dimensional space, with one dimension being the value of the card in the dealer’s hand, ranging from ace to 10, the sum of the values of the player’s cards, ranging from 12 to 21, and the existence or not of a usable ace in the player’s hand, for a total of 200 states. The agent has two available actions: hit, represented by a 1, and stick, represented by a 0.

The optimal policy for a simple game of blackjack with only hit and stick actions was defined in [3], and is shown in Figure 2. This policy may be used as a benchmark for the trained algorithms.

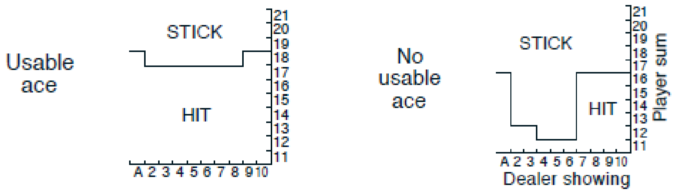


Figure 2 – Optimal policy for an agent playing blackjack, with or without usable ace, respectively

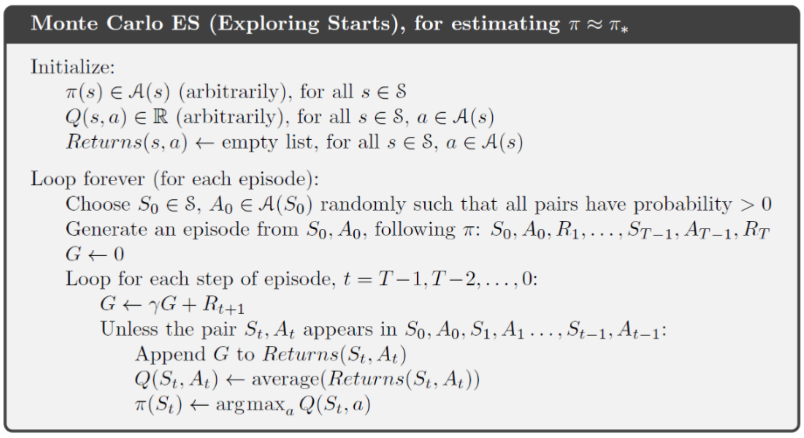
4. Monte Carlo methods

Monte Carlo methods, when applied to reinforcement learning problems, have the advantage of not requiring complete knowledge of the environment, namely the probabilities of getting a certain reward when starting in a certain state. For example, the problem of determining the probability of winning given a certain hand and dealer card is difficult to solve, making Monte Carlo an appropriate technique to deal with this problem, since the model used only simulates transitions between states and calculates rewards, with no calculation of probabilities required. Blackjack is also an episodic (every game is an episode) and terminating environment, meaning these algorithms are possible to apply as defined in [1].

Monte Carlo methods are based on averaging the returns over all samples given a large enough number of samples, for each state-action pair. For the average of the samples to converge to the optimal policy, an infinite number of iterations would be necessary but in practice this constraint is relaxed by aiming for an approximation of the optimal policy up to a certain magnitude of the error. This is done by using the returns after each episode to evaluate the policy, followed by improvement of the policy in the states visited in the episode.

Another issue that Monte Carlo Control (the approximation of optimal policies) deals with is maintaining explorations. There are two ways to deal with this problem, *with* or *without exploring starts*.

With exploring starts, the algorithm starts in a state-action pair chosen at random from the state-action space, guaranteeing that every pair as a non-zero probability of being chosen. With enough iterations, every pair should be chosen at least once. Figure 3 shows the pseudocode used as a basis for this method, where the only change was saving only the number of visits and mean for each state instead of the whole Returns list, to save memory.



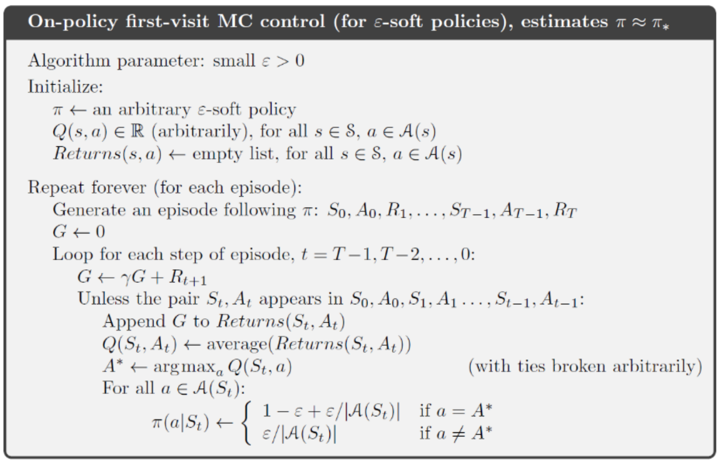
```
Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
    $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability $> 0$
    Generate an episode from $S_0, A_0$, following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
            $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$
```

Figure 3 – Monte Carlo Exploring Starts Control pseudocode

While this method may be possible to apply to blackjack due to its small state-action space, exploring starts are difficult to implement in more complex problems due to the difficulty in generating a model that can initialize at any state. Instead, an $\epsilon - greedy$ on-policy control method is used in those cases. On-policy method means that the process will improve upon the policy used to make the decisions, while $\epsilon - greedy$ means that the policy is to choose the greedy action (the one that leads to maximum estimated value) most of the time, with a probability $1 - \epsilon$, while more infrequently, with probability ϵ, it will choose at random. In an environment like blackjack, where there are only two actions, instead of choosing at random the exploratory actions just consist of choosing the action opposite to the one in the policy. Figure 4 shows the pseudocode used to implement this method.



```
On-policy first-visit MC control (for $\varepsilon$-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
            $A^* \leftarrow \text{argmax}_a Q(S_t, a)$                    (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

Figure 4 - Monte Carlo $\epsilon - greedy$ Control pseudocode

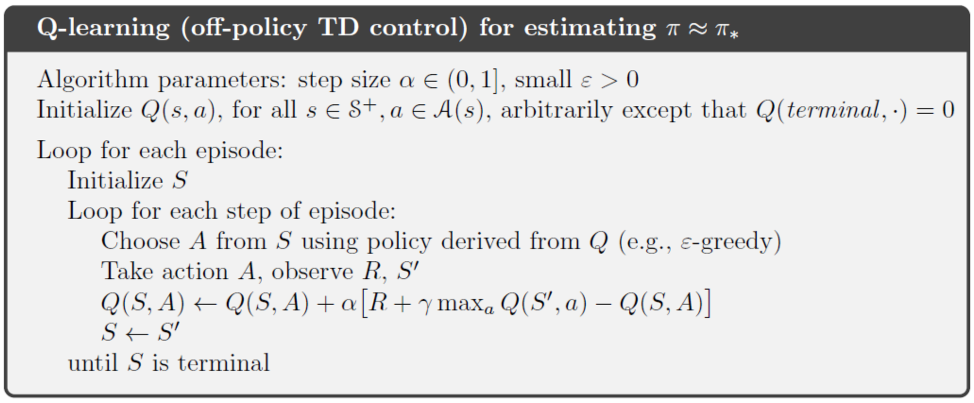
5. Q-Learning

Q-Learning is a type of *Temporal Difference* learning algorithm. While this type of algorithms shares the ability of Monte Carlo of learning from experience without a direct probabilistic model of the environment, unlike the previous method they can learn and update their estimates without waiting for the end of the episode, updating instead at every time step.

Q-Learning was developed by [4], and is defined by the following update formula:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{5}$$

Note that in this case, the action-value function is updated and converges to the optimal action-value function q^*, independently of the followed policy. This means that this is an *off-policy* control method, where the learned optimal policy, called the *target policy*, and the policy used in the learning process, called the *behaviour policy* are different. To converge to the optimal policy, the only requirement is the updating of all state-action pairs, which should be guaranteed up to an arbitrary error with enough iterations. In this work's implementation of this algorithm, the policy used as behaviour policy was $\epsilon - greedy$, as described in the section about Monte Carlo methods. Figure 5 shows the pseudocode for the Q-Learning method.



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Figure 5 – Q-learning pseudocode

5.1. ε-decay

One problem with using an $\epsilon - greedy$ policy for Q-Learning is the fact that keeping the probability of taking an exploration action ϵ is kept constant throughout the training process. This means that, even when the agent's policy is close to convergence to the optimal policy, it still takes exploration actions too often, worsening performance. Two solutions to this problem were implemented: exponential ϵ decay and Boltzmann exploration.

Exponential ϵ decay is a simple procedure: every iteration, the value of ϵ is multiplied by a value between 0 and 1, meaning that it is decreased over time during the learning process, tending to 0 with infinite iterations.

Boltzmann exploration [5] is another method of choosing the action, where the probability of choosing an action is given by a Boltzmann distribution:

$$\Pr(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{a'} e^{\frac{Q(s,a')}{T}}} \tag{6}$$

In the above equation T is a temperature parameter that decays exponentially over time, yielding a similar effect to the exponential ϵ decay, where off-policy exploration actions are favoured at the start of the learning process and become less likely with each iteration. The way the probability distribution is calculated favours off-policy actions with high Q-value while making clearly low value actions more unlikely, with the drawback of higher computational cost at every iteration.

6. Deep Q-Learning

Q-learning raises a problem when dealing with large, state-action spaces, both in terms of computing power and memory required to store the Q matrix containing the correspondence between all states and actions. As such, one solution to this problem is the approximation of the Q-table using an Artificial Neural Network, since this type of structure is a universal non-linear approximator [6]. Neural Networks also provide advantages due to their ability to learn complex patterns and generalize to cases it has never been exposed to before (when exposed to a new case, the Q-Learning method will take a random action instead). This method is called Deep Q-Learning (DQL), due to the common use of Deep Neural Networks to approximate the Q Table.

Figure 6 shows the difference between the Q-Learning and the Deep Q-Learning algorithms. While in Q-Learning a table is stored with every state-action pair as inputs and corresponding Q-value as output, while in DQL the state is given as an input to the Neural Network, that approximates the Q Table and outputs the Q-Values of every possible action. The optimal action chosen is then the one with the maximum Q-value that is produced by the network.

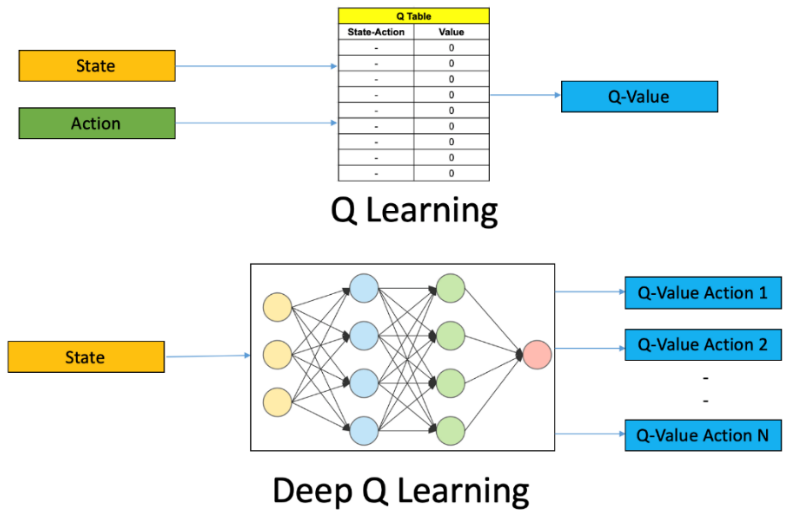


Figure 6 – Comparison between Q-Learning and Deep Q-Learning

For a network to be trained, a target must be defined. Reusing equation 5 that defines the updates of the Q Table, the target may be set to $R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a)$. This presents another problem, since the recurrency of the relation means the target will change at every iteration, which does not happen in traditional Deep Learning where the target is fixed, creating non-stationarity and instability, and making convergence much more difficult to attain. The solution is to implement a second neural network with fixed parameters, called the *target network*, whose objective is to estimate the target. This target network as the same structure as the network responsible for approximating the Q Table, called the *prediction network*. Every given number of iterations, the weights of the prediction network are copied to the target network, generating a new target, as shown in Figure 7.

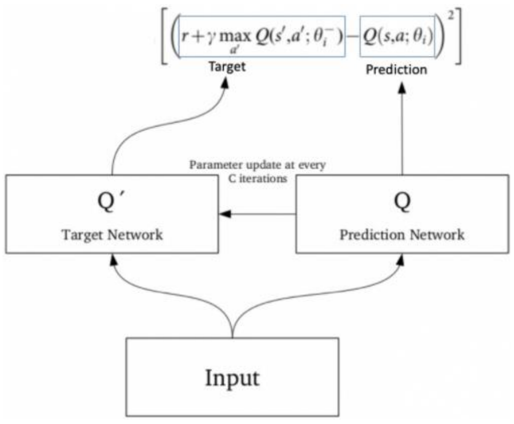

$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

Figure 7 – Interaction between target and prediction network to approximate the Q table

Another precaution taken to increase stability is instead of training the prediction network at every episode, which would be prone to overfitting, past values of states and corresponding Q-values are stored in memory, from which a minibatch is randomly taken to train the prediction network. The way data is generated for minibatch sampling is, as discussed before in Q-Learning, with an $\epsilon - greedy$ policy with exponential decay of ϵ.

The code for this section was adapted from [7].

7. Algorithm Analysis and Results

7.1. Monte Carlo

7.1.1. Exploring Starts

The Monte Carlo Control with Exploring Starts is the simplest method implemented for the resolution of our problem. Having no parameters to be modified, it ran one time with 10 000 000 iterations. The final policy and value function for each evaluated state can be seen in Figure 8. The number of final suboptimal actions in the estimated policy (compared to the optimal solution in Figure 2) and its evolution through every 10 000 iterations can be seen in Figure 9. The final number of suboptimal actions was 12 out of 200.

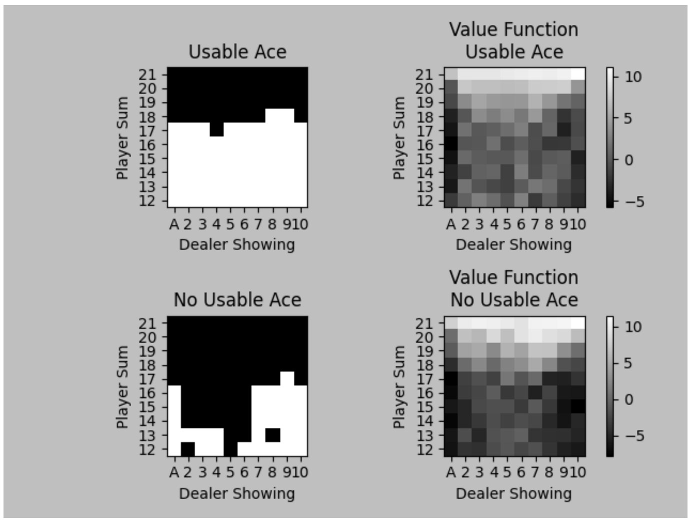


Figure 8 - Policy and Value Functions for MC Exploring Starts (iterations = 10 000 000)

Our approach to the problem dictated that in the state corresponding to a player hand lower than 12, the simulation would just hit for a new card, and the state is not considered. This can be done as in those states the correct action is always to hit because it is impossible to lose by going bust. This type of formulation aids significantly in this method, as well as all other methods considered, as it makes it faster and more likely to generate any state as a first state thus allowing the exploration for every state as a starting state.

As a result of this, the Monte Carlo Control with Exploring Starts proves to be highly successful because, even though it takes the least computational time, this method outputs some of the best results of the whole report evidenced by the small number of suboptimal actions. Further validation tests were executed, and the final number of suboptimal actions were very similar, always around 11-14, validating these results.

Finally, it is important to note that the evolution of the number of suboptimal actions in Figure 9 seems to indicate that the number of suboptimal actions would continue to decrease if more iterations were performed, meaning that this method would possibly reach an optimal solution given enough iterations.

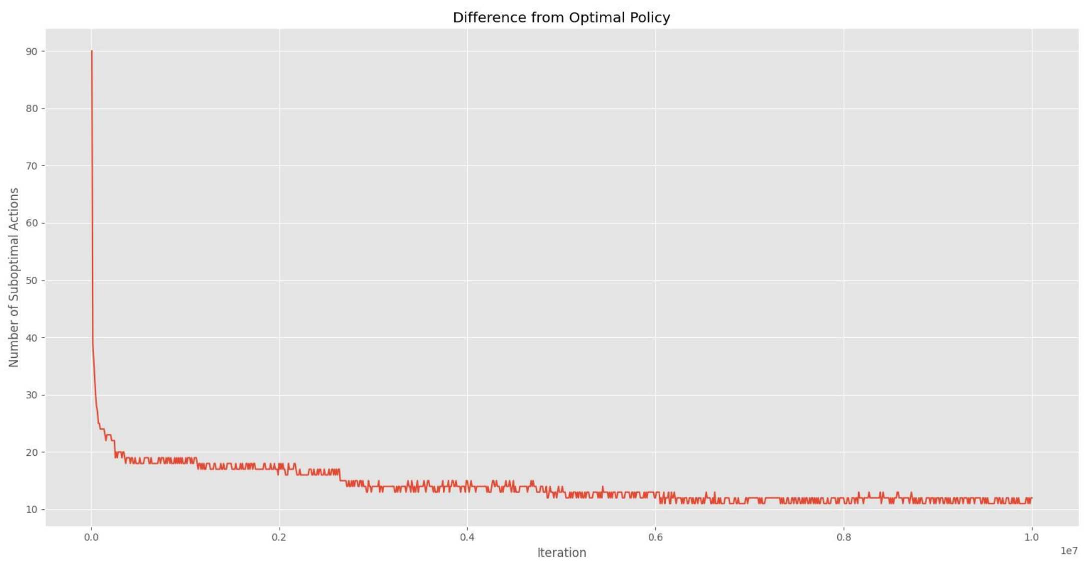


Figure 9 - Suboptimal Actions Evolution for MC Exploring Starts (iterations = 10 000 000)

7.1.2. Without Exploring Starts

The Monte Carlo Control Without Exploring Starts uses an $\epsilon - greedy$ policy, where ϵ is the probability of the simulation taking a suboptimal action for sake of exploration. As mentioned in section 5.1, using a constant ϵ may not be optimal, so the three different methods for finding the value of ϵ were tested with three different parameters each. Each configuration ran 5 times for 10 000 000 iterations and the average win rate and suboptimal actions are shown in Table 1. It is important to note that some minimum values of ϵ (=0.001) and Temperature (=0.1) were implemented to ensure that the ϵ value is not too low. This is done because if the ϵ value is too low, there is virtually no exploration and subsequently, no improvement to the policy is possible.

	ε-greedy			ε-decay			Boltzmann Exploration		
	Value of ε			ε Decay Rate			Temperature Decay Rate		
	0.05	0.10	0.15	1%	0.1%	0.001%	1%	0.1%	0.01%
Win Rate	42,51	42,32	42,58	42,34	42,33	42,39	42,44	42,66	42,61
Suboptimal Actions	24,5	27,4	23,3	27,1	29,6	28,4	25,4	19,7	24,8

Table 1 - MC Without Exploring Starts Win Rate and Suboptimal Actions

The Boltzmann Exploration, with a Temperature Decay Rate of 0.999 is the simulation with the lowest suboptimal actions, but comparing to the Monte Carlo Control with Exploring States, this solution is unsatisfactory, as it is computationally slower, and the number of suboptimal actions is higher. We can conclude that Monte Carlo Control Without Exploring States is not the best fit for this blackjack problem. This method would be better suited in a more complex reinforcement learning problem where the starting states are difficult to generate, and the number of possible actions is greater.

7.2. Q-Learning

The Q-Learning algorithm was implemented with three different exploratory policies: $\epsilon - greedy$, exponential $\epsilon - decay$ and Boltzmann exploration. Before comparing the performance of each policy, we tested the influence of the step size α in the algorithm. Using a $\epsilon - greedy$ exploratory policy with $\epsilon = 0.1$, the algorithm ran for 10 million iterations with 4 different values of step size ranging between 0.025 and 0.1. At every 100 000 iterations, the best policy based on the Q-table was derived by taking the action with highest value at each state. This policy was then tested on 10 000 games and the win rate was recorded. The results are shown in Figure 10. Additionally, since we have access to the optimal policy, we can also record the number of suboptimal actions. The results are shown in Figure 11.

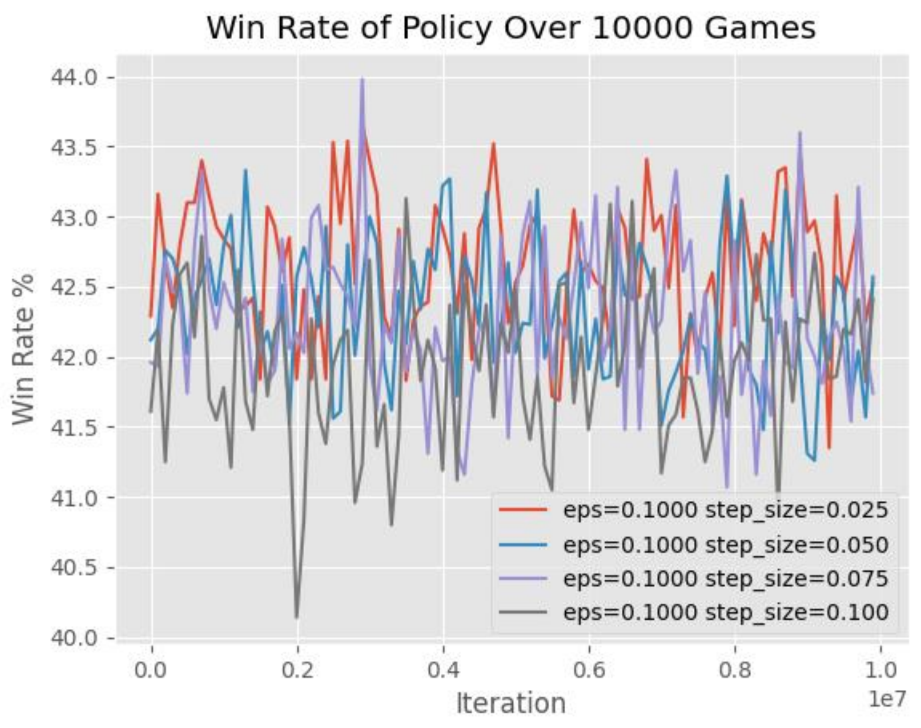


Figure 10 - Influence of step size in Q-Learning: win rate of policy over 10 000 games

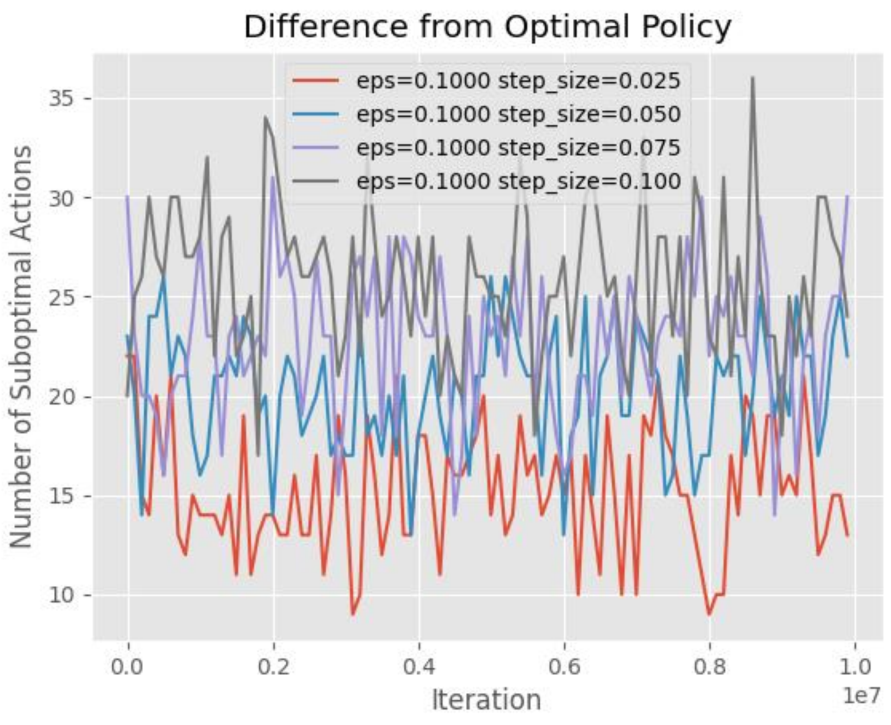


Figure 11 - Influence of step size in Q-Learning: number of suboptimal actions

The results are clear, a lower step size results in a better win rate and a better policy. A reason for this might be the stability of the algorithm. As the environment is probabilistic, the optimal action is not guaranteed to lead to a positive reward and in the case of a loss, the update on the Q-table will be detrimental to the overall solution. A large step size will lead to substantial changes in the Q-table and for states where both actions have very similar expected returns, the algorithm will have difficulty converging. Therefore, a lower step size will improve stability by making less drastic changes on the Q-table.

To test the performance of each exploratory policy, 5 runs of 10 million iterations were performed. At the end of training, the policy was derived from the Q-table and tested on 1 million games. The average win rate and the average number of suboptimal actions for the 5 runs are presented in Table 2. $\epsilon - decay$ was implemented with a minimum ϵ value of 0.001 and Boltzmann Exploration was implemented with a minimum temperature value of 0.1.

	ε-greedy			**ε-decay**			**Boltzmann Exploration**		
	Value of ε			ε Decay Rate			Temperature Decay Rate		
	0.05	0.10	0.15	1%	0.1 %	0.01%	1%	0.1 %	0.01 %
Win Rate	42,39	42,57	42,47	42,46	42,50	42,35	42,84	42,62	42,59
Suboptimal Actions	16,6	17,8	16,2	18,2	18,6	20,8	12	16	15,6

Table 2 – Q-Learning average results for 5 runs of 10 million iterations.

The results show that using Boltzmann Exploration as an exploration policy leads to better results. Boltzmann Exploration chooses actions with probability based on the values of Q-table and as such, for states where one action has a much larger expected return, that action is more consistently picked than with constant ϵ methods which lead to a reduction in the exploration of suboptimal paths. The temperature decay ensures that the algorithm does not converge to a suboptimal solution prematurely and promotes more exploration in earlier iterations.

7.3. Deep Q-Learning

Deep Q-Learning was implemented using a neural network from the Keras API in Python. Due to the already low dimensionality of the input space (dealer card, sum in hand and useable ace – three dimensions), the typical Convolutional Neural Net, that extracts featured from high dimensional data in convolutional layers and then reduces dimensionality in max pooling layers. A simple architecture using two dense (fully connected layers) with ReLU activation functions was used, with 10 neurons per layer.

Three hyperparameters were studied: the exponential decay rate of ϵ, the training minibatch size and the target update interval (interval, in number of iterations, between target neural net updates).

For the $\epsilon - greedy$ policy, an exponential $\epsilon - decay$ policy with a decay of 0.99995 or 0.9995 was followed, for the batch size 32, 64 and 128 games were used and for the target update interval values of 10 and 20 were tested. The model was trained for 25000 iterations for all models. Better results could be obtained with more iterations, but since at this point the algorithm takes over 1 hour to run, for practical reasons it was not increased further. The quality of the model was measured using two metrics: the win rate over 5000 games of the trained model (in percentage) and the number of actions different from the optimal policy. The results are shown in Table 3.

Model Number	ε decay rate (%)	Minibatch size	Target update interval	Win rate (%)	Number of suboptimal actions
1	0.005	64	10	42.76	15
2	0.05	64	10	42.98	23
3	0.005	128	10	42.37	23
4	0.005	64	20	41.26	29
5	0.005	64	15	42.76	10
6	0.005	32	15	41.56	20

Table 3 – DQL results for various hyperparameter selections

The best result was obtained for model number 5, with a 64 game minibatch, 0.005% decay rate and a 15-iteration target update interval. Win rate did not vary significantly between models but it was the one with the closest policy to the optimal. The resulting policy is shown in Figure 12, along with the value function. As expected, the value function is much higher when the player has a high sum in hand (18 to 21). Also note the lower value when the dealer has an ace, due to the high likelihood of getting either an instant natural, which guarantees a draw or drawing a high card without going bust.

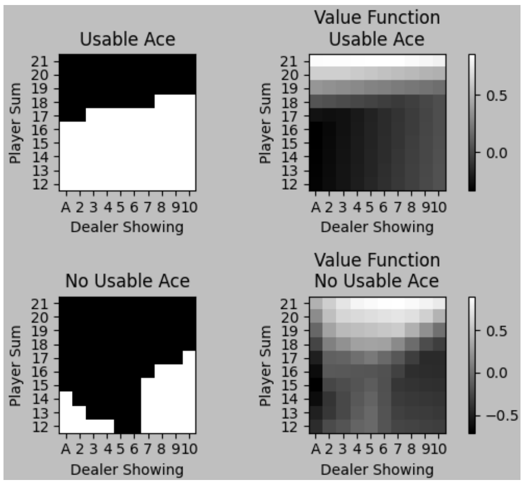


Figure 12 – Policy and value function of the best DQL model

8. Method comparison

Looking at all the methods implemented, Monte Carlo with Exploring Starts proved to be the best one, since it got the highest performance in terms of approximation of the optimal policy in general, only being surpassed by one of the Deep Q-Learning models. Note that this is only possible due to the low dimensionality of the state-action space, and the ability of the model to produce all possible states with non-zero probability in a realistic number of iterations.

Q-Learning provided mostly better results than Monte Carlo without Exploring Starts in terms of similarity to the optimal policy, with $\epsilon - greedy$ exploration having better results than exponential decay in both methods, and since this second ,method requires more computations than using a constant ϵ, there is no advantage in using it for this problem. Boltzmann exploration outperformed both previous methods though, justifying its added computational cost.

Deep Q-Learning was able to approximate the optimal policy as the previous two methods, but the added computation time showed that this method is too complex for the blackjack problem and should only be applied in a more complex reinforcement learning environment.

The win rate did not vary much when changing method, exploration strategy or hyperparameters, showing that as long as the agent learns a good basic strategy, the long-term success is not too dependent on the slight variations in policy.

9. Implementation

The Python implementation of all the algorithms demonstrated is available in the following link:

https://github.com/BjornIronside/SInt_FinalProject.git

Available scripts:

- **QLearning.py** – Implements the Q-Learning algorithm with the three exploration methods presented, as well as some result visualizations.
- **blackjack.py** – Contains the Gym blackjack environment.
- **deepq.py** – Implements the Q-Learning algorithm with the three exploration methods presented, again with visualization.
- **montecarlo.py** – Implements both Monte Carlo methods, and corresponding plots.
- **visualization.py** – Contains some functions used for results analysis and visualization.

10. Conclusion and Future Work

In this work, three types of algorithms were applied to the Blackjack problem: Monte Carlo, Q-Learning and Deep Q-Learning.

All three methods converged to a policy that was suboptimal but close to the optimal policy, with only 10 to 20 actions in 200-element state-action space being different from the proposed the optimal solution. The win rate was always between 40 and 43 %, showing the well-known fact that the house (casino or dealer) always has advantage in the long term.

The Deep Q-Learning algorithm was much slower compared to the other two, and the problem is not complex enough to justify its application due to memory constraints. This was further complicated by the fact that no computer was available with GPU acceleration, which would have accelerated the calculations.

As future work, it would be relevant to see how well each algorithm performs for added complexity to the environment: adding extra actions such as doubling down and splitting, taking advantage of a finite deck, and trying to get the algorithm to learn an equivalent to card counting and allowing for variable bets, where the agent would be able to adjust its bet according to the state of the environment. All these additions would hopefully lead to an algorithm that gets a win rate closer to 50%, as would be expected from a professional human blackjack player.

Another potential area of improvement is the experimentation with policies that estimate the benefit of exploration using the classical notion of Value of Information – the expected improvement in future decision quality that might arise from the information acquired by exploration [5].

11. References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning, Second Edition: An Introduction - Complete Draft*. 2018.

[2] G. Brockman *et al.*, “OpenAI Gym,” Jun. 2016, Accessed: Dec. 09, 2020. [Online]. Available: https://arxiv.org/abs/1606.01540.

[3] D. R. Brillinger and E. O. Thorp, “Beat the Dealer: A Winning Strategy for the Game of Twenty One.,” *Am. Math. Mon.*, vol. 72, no. 4, p. 438, 1965, doi: 10.2307/2313530.

[4] C. J. C. H. Watkins, “Learning from delayed rewards,” *Robotics and Autonomous Systems*, vol. 15, no. 4. pp. 233–235, 1989.

[5] R. Dearden, N. Friedman, and S. Russell, “Bayesian Q-learning,” *Proc. Natl. Conf. Artif. Intell.*, pp. 761–768, 1998.

[6] G. Cybenko, “Approx_By_Superposition.Pdf,” *Mathematics of Control, Signals, and Systems*, vol. 2. pp. 303–314, 1989.

[7] “Deep Q Learning and Deep Q Networks (DQN).” https://pythonprogramming.net/deep-q-learning-dqn-reinforcement-learning-python-tutorial/ (accessed Dec. 10, 2020).