ML Project Report

Gianluca Sanfilippo 1943663

Artificial Intelligence & Machine Learning – Eng. C.S. – Sapienza University of Rome

Reinforcement Learning Project based on Q-Learning algorithms

1 Introduction

Reinforcement Learning (RL) is a type of Machine Learning (ML) where an Agent learns the actions leading to the best result by interacting with the Environment.
In fact, while in other approaches such as Supervised Learning, the agent has samples of the form **(state, policy_for_state)**, in RL the goal is to find the aforementioned policy, which is not given a priori but learned by executing actions in the environment, collecting the reward (positive/negative) and therefore reaching the successive state.

1.1 MDP and Reinforcement Learning

As depicted before, obtaining the policy is the ultimate goal of an RL agent.
To formalize the problem and therefore the solution of a Reinforcement Learning scenario, we introduce the Markov Decision Process (MDP), which is described as a tuple

$$\mathbf{M = (S,\ A,\ \delta,\ r)}$$

S is the finite set of states the agent can be

A is the finite set of actions the agent can perform

$\delta(s' \mid s, a)$ is the probability distribution function over transitions **s** to **s'** by performing action **a**

$r : S \times S \to \mathbb{R}$ is the reward function over transitions, used as r(s,a)

Given this setting, the environment in which the RL agent performs is completely defined. In fact, by the current state **s**, the agent performs an action **a** and collects the associated reward **r**, ends up in a new state **s'** (or stays in the

same state according to the transition function).
With such information, the model can learn the best action to take for each state in which it is in, which means building the ***policy function***.

1.2 V-Function and Q-Function

The policy function $\pi : S \to A$ returns the action to take for each state, and since the goal is to maximize the expected reward of each action for each given state, such a policy should be built according to this objective.
To obtain such a constructed policy, two fundamental functions of Reinforcement Learning must be introduced.

- **V-Function** (state value function):
  $V^{\pi}(s) \equiv \mathbb{E}\big[r_1 + \gamma r_2 + \gamma^2 r_3 + ...\big]$ where $\gamma \in [0, 1)$ is the discount factor and **s** is the state from which the rewards r_i are calculated and collected.
  The V function is also called Expected Cumulative Discounted Reward.
  Note that for a deterministic environment, the rewards are not random variables and therefore the function assumes the form $V^{\pi}(s) \equiv r_1 + \gamma r_2 + \gamma r_3 + ...$ since the rewards are defined constant.
  $V^{\pi}(s)$ returns the expected reward (or value) obtained by applying policy π in state s.

  Therefore, the solution to the MDP becomes finding the optimal policy π^* (Π) such that $V^{\Pi}(s) \geq V^{\pi}(s)\ \forall s$.

- **Q-Function** (state-action value function):
  $Q^{\pi}(s, a) = r(s, a) + \gamma V^{\pi}(\delta(s, a))$ which means
  $Q^*(s, a) = r(s, a) + \gamma \max_{a' \in A}\{Q(\delta(s, a), a')\}$
  More generally, $Q(s, a) = \mathbb{E}\big[r(s, a, s') + \gamma V^*(\delta(s' \mid s, a))\big]$
  $Q^{\pi}(s, a)$ expresses the expected utility (value) of executing action a in state s and then acting according to π.
  We will show how maximizing the Q function for every state s means building the optimal policy π^*.

1.3 Q-Learning approach

In the previous sections we have laid the foundations of the algorithms we use in this project, which are based on learning the maximum Q value for each state s.
In fact, as we anticipated in the introduction section, the environment of a reinforcement learning model does not give us the reward and the transition functions, which must be discovered and learned to progressively optimize our policy, and thus learn how to best act in the environment.

In this project we will see two algorithms: **Tabular Q-Learning** and **Deep Q-Learning Neural Network (DQN)**
Both algorithms are based on the Bellman update equation for Reinforcement Learning, which is listed below.
$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)\big]$

2 The Blackjack Game

The project we are presenting is based on the blackjack game, intrinsically stochastic and non deterministic, since the game itself is by design highly probabilistic and made to have an average losing pattern for the player.

Blackjack is a card game where the goal is to beat the dealer by obtaining cards that sum to closer to 21 (without going over 21) than the dealers cards. The game starts with the dealer having one face up and one face down card, while the player has two face up cards. All cards are drawn from an infinite deck (i.e. with replacement).

The card values are:

- Face cards (Jack, Queen, King) have a point value of 10.
- Aces can either count as 11 (called a 'usable ace') or 1.
- Numerical cards (2-10) have a value equal to their number.

The player has the sum of cards held. The player can request additional cards (hit) until they decide to stop (stick) or exceed 21 (bust, immediate loss). After the player sticks, the dealer reveals their facedown card, and draws cards until their sum is 17 or greater. If the dealer goes bust, the player wins.

If neither the player nor the dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21.

For this game we are implementing the environment "Blackjack-v1" provided by Gymnasium, one of the most famous open source framework providing a large amount of environments for RL projects.

Gymnasium environments are designed to provide interfaces with which is possible to interact with the environment by performing actions, collecting the corresponding rewards, next states and observations step by step until reaching a Done flag, which means the "episode" we are in is terminated.

Thanks to the feedback from the environment, specifically, for a given action-state pair, the reward and the observation alongside to the resulting new state provided, allow the RL agent to update and therefore learn and optimize the policy.

The action space for this environment is the following:

- **0 (stick)**
- **1 (hit)**

The observation space is of the form;
(int(), int(), int())
where the numbers respectively represent:

- the player's current sum,
- the value of the dealer's one showing card (1-10 where 1 is ace),
- whether the player holds a usable ace (0 or 1).

The possible rewards are:

- **+1 (win game)**
- **0 (draw game)**
- **-1 (lose game)**

The episode ends if the following happens:

- The player hits and the sum of hand exceeds 21.
- The player sticks.

An ace will always be counted as usable (11) unless it busts the player.

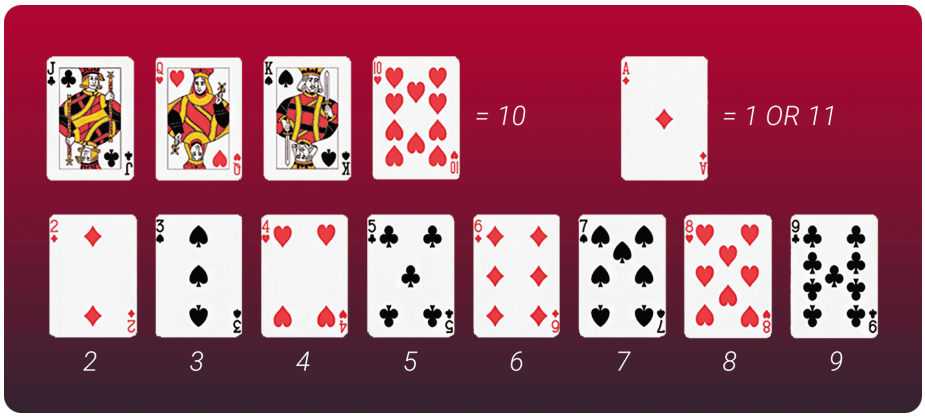


Figure 1: Blackjack card values

3 Implementation

The Blackjack environment depicted before and implemented in this project is entirely **non deterministic**. Although it is possible to build it deterministic thanks to some environment settings, it is considered non realistic since the nature of the game itself is probabilistic.
Thus we used the Non Deterministic update equation as described before.

3.1 Tabular Q-Learning

In Tabular Q-learning, the Q function is built iteratively starting from all zeroes and, for each action taken, it is updated with the best value gained from such action or the previous best one if greater.

For what is concerned the action to perform at each step, it is carefully chosen among two possible options according to a famous strategy.
The $\epsilon - greedy$ strategy:
thanks to the ϵ parameter, choose best action with probability $1 - \epsilon$, choose random action with probability ϵ.
The first option is called *exploitation* and, to be practical, needs knowledge about what is the best thing to to in a certain state.
The second one is called *exploration* and is the unique choice at the first rounds when such mentioned knowledge is not yet available to the agent.
To first prioritize exploration and then, in the successive steps, when knowledge is further available as the Q table gets optimized, prioritize exploitation, the ϵ parameter decays with a factor (in this case the epsilon decay is constant over each episode until it reaches a threshold, after which it does not changes until

the end of the execution).
Here is shown the implementation of the epsilon decay function.

```
    # epsilon-greedy strategy
    def choose_action(self, s: State) -> Action:
        #random action
        if random.random() < self.epsilon:
            return random.choice(self.actions)
        #best action
        qs = self.get_qs(s)
        # break ties randomly in case multiple actions yeld max
    return
        maxv = np.max(qs)
        candidates = [a for a, q in enumerate(qs) if q == maxv]
        return random.choice(candidates)
    def decay_epsilon(self):
        self.epsilon = max(self.epsilon_min, self.epsilon * self.
    epsilon_decay)
```

Below is shown a snippet of the code in which the aforementioned update is performed.

```
    def update(self, s: State, a: Action, r: float, s2: State, done
    : bool):
        q = self.get_qs(s)[a]
        if done:
            target = r
        else:
            target = r + self.gamma * np.max(self.get_qs(s2))
        td = target - q
        self.get_qs(s)[a] += self.alpha * td
###
    for episode in episodes:
        while not done:
            a = agent.choose_action(s)
            obs2, r, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            agent.update(s, a, r, s2, done)
```

While advancing with the training during episodes, stats about rewards, wins, losses, draws, epsilon updates.
Thanks to this stats we are able to show in the Results section useful information about the outcomes of the training varying the parameters, specifically: α (learning rate), ϵ, ϵ_decay, γ, number_of_episodes.
Thanks to such parameters we are able to tune the model to better approximate the best policy for the given environment.

3.2 Deep Q-Learning (DQN)

In DQN approach, the Q table is approximated by using a Neural Network.
Such N.N. transform the table from a discrete space to a continuous space and

allows to learn the Q function by liner regression.
An advantage with respect to tabular Q learning approach is that yet unseen states, thanks to the non discrete environment, output a non zero value and, therefore, the table is still able to guide the agent into new environment states by applying non totally random action choices.

$Q_\theta(s, a) = \theta_0 + \theta_1 F_1(s, a) + ... + \theta_n F_n(s, a)$

The result of a DQN elaboration is the optimization of such continuous Q function minimizing the loss function through experiments so that for each state prompted as input to the N.N., it outputs real values corresponding to the best action to take, which is the action yielding the maximum reward (value) for the given state.

In this project we implemented a Deep Neural Network with 2 fully connected hidden layers. Each layer is composed of 32 activation units with ReLU non linear functions.
The network also implements a replay buffer of adjustable length in which are stored past transitions of the form *(state, action, reward, next_state)*.
This allows the agent to use past experience to update the network during training by randomly selecting minibatches of this buffer. The size of such minibatches is again adjustable.

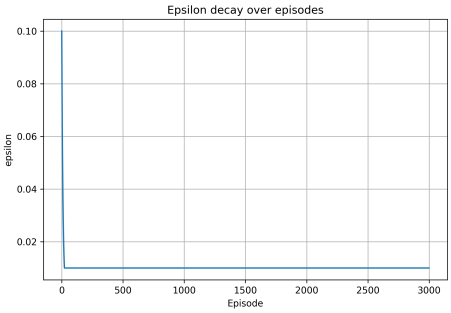
Like for the tabular approach, the same statistics have been collected during training, thanks to which we have been able to compare the agent performance varying the hyperparameters.

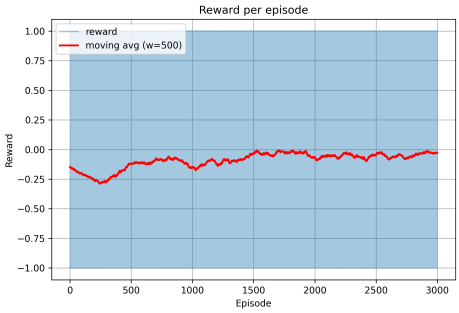
4 Results

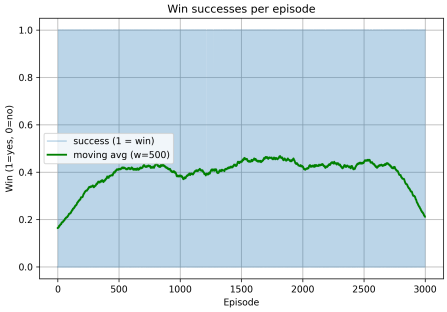
4.1 Non Deterministic - Tabular Q-Learning

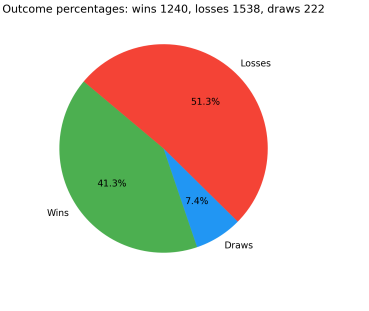
Here are shown the plots created from the training process.
we focused on the reward per episode, which gives an understanding on the trend of the training process together with the successful episodes and the pie graph showing the win/loss/draw percentages.

alpha=0.1, epsilon:0.1 - 0.01, epsilon_decay=0.9, gamma=0.9, episodes=3000

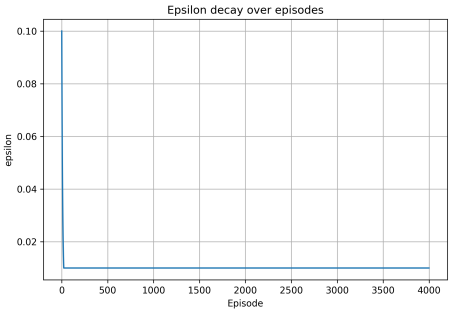


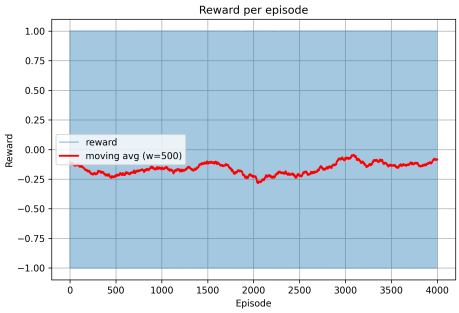


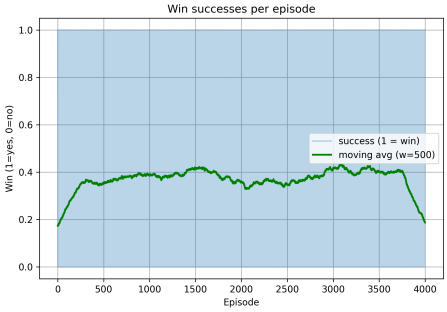


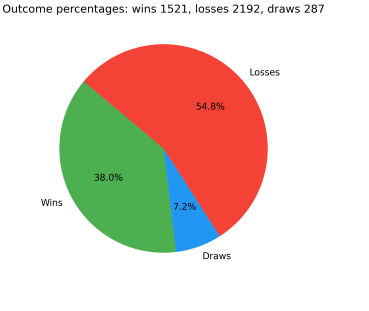


alpha=0.1, epsilon:0.1 - 0.01, epsilon_decay=0.9, gamma=0.9, episodes=4000

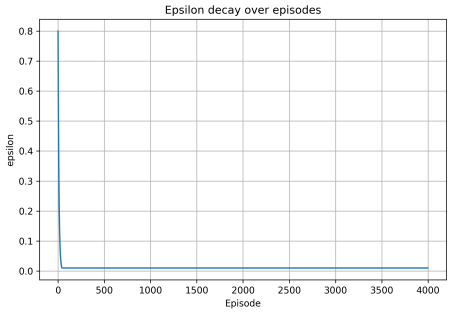


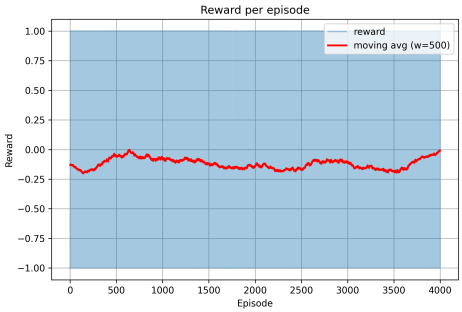


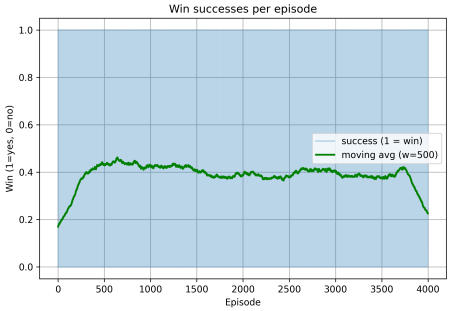




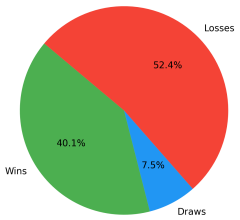
alpha=0.2, epsilon:0.8 - 0.01, epsilon_decay=0.9, gamma=0.9, episodes=4000



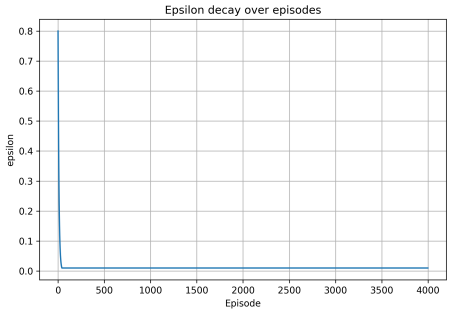


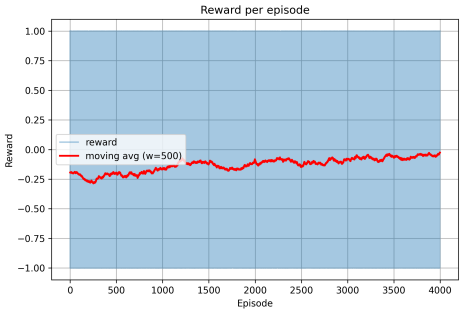


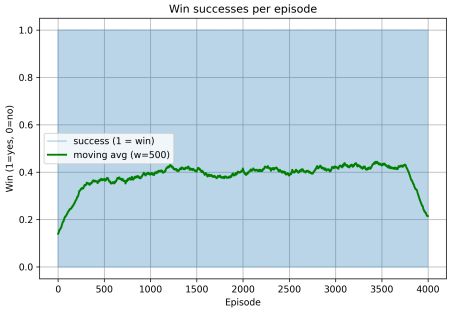
Outcome percentages: wins 1604, losses 2097, draws 299



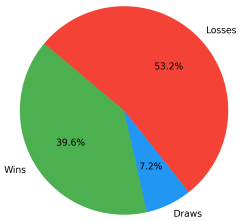
alpha=0.3, epsilon:0.8 - 0.01, epsilon_decay=0.9, gamma=0.9, episodes=4000

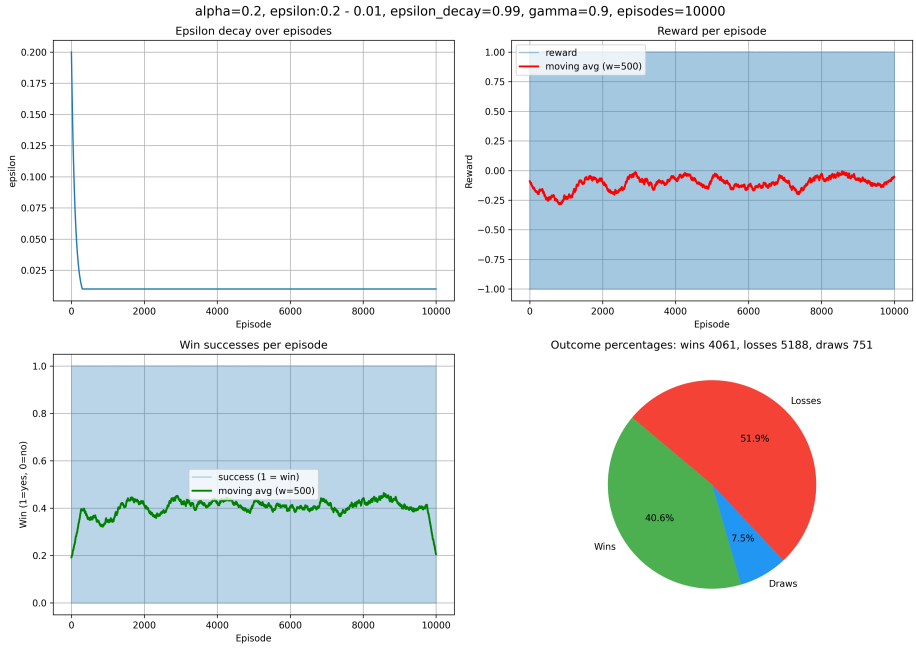


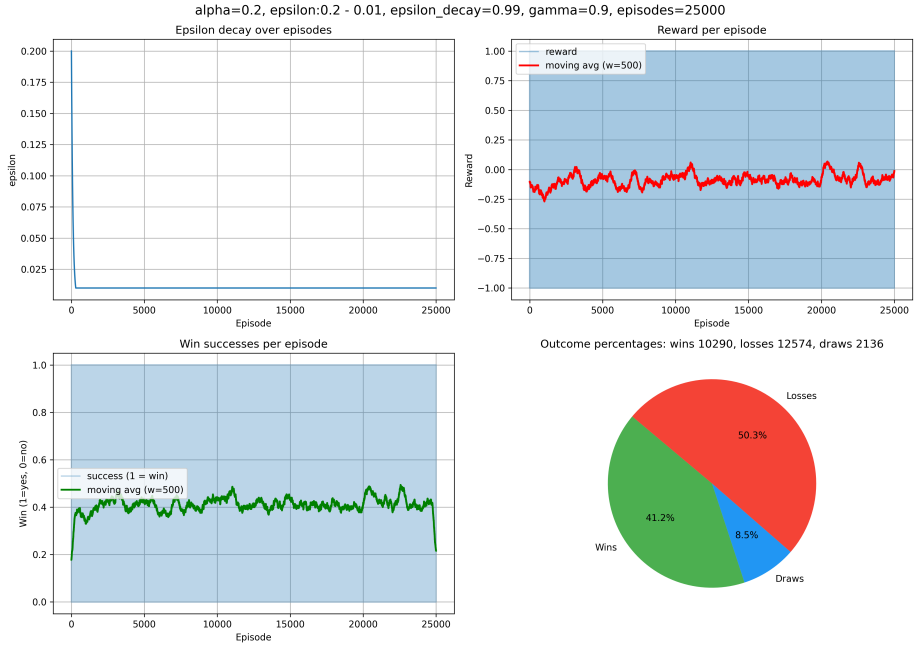




Outcome percentages: wins 1585, losses 2128, draws 287

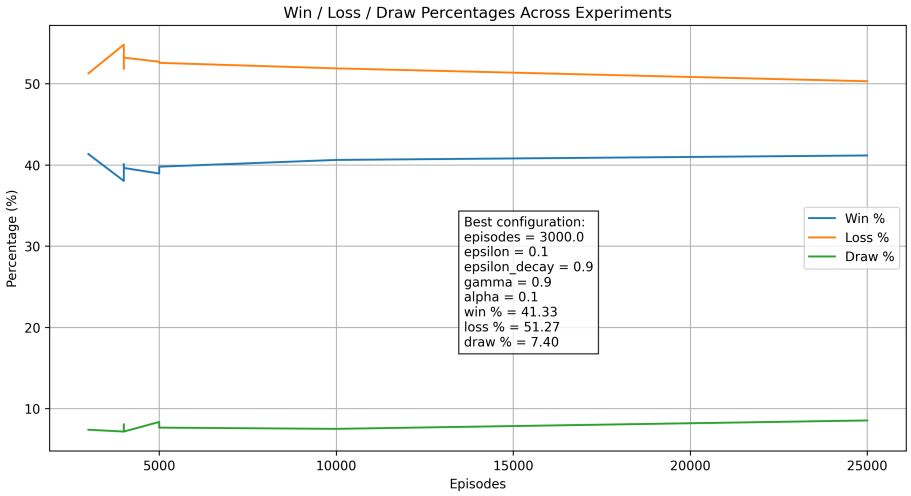






To evaluate such results, we have implemented a script which analyzes the outcomes of the training in the settings just exposed, and obtain the best one

among them by maximizing the win percentage and minimizing the loss percentage.
The final result of the analysis is depicted in the last picture of this section.

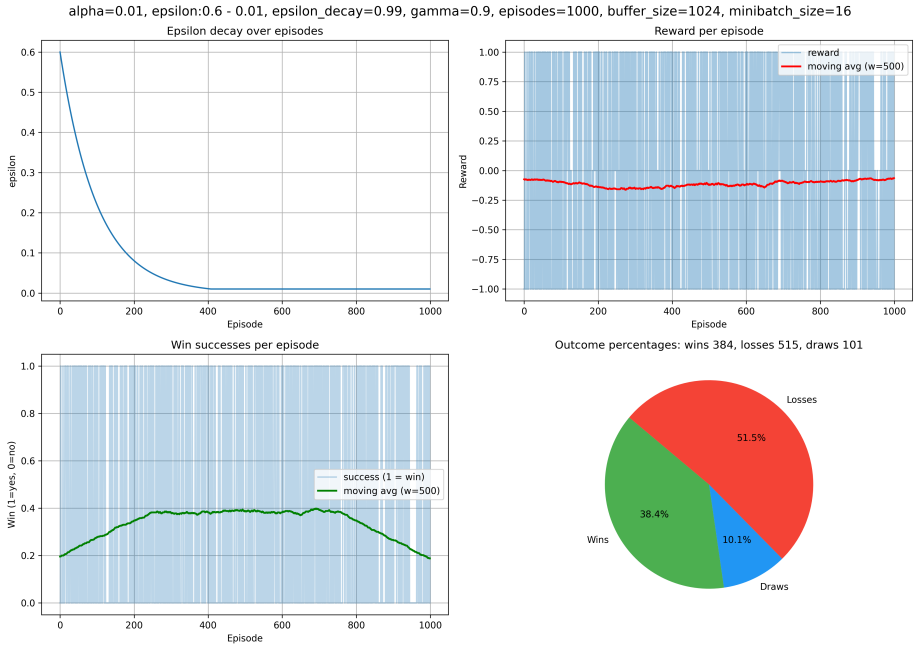


The best setting of parameters has been identified in the first one; even though such setting has the minimum number of episodes, it outperformed the runs with much more episodes in terms of minimum losses.

4.2 Non Deterministic - Deep Q-Learning (DQN)

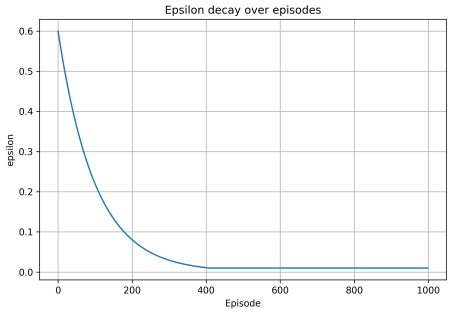
At first, we tried with the following hyperparameters:
number of episodes = 1000
$\alpha = 0.001$
$\gamma = 0.9$
$\epsilon = 0.6$
$\epsilon decay = 0.99$
replay buffer length = 1024
minibatch size = 16

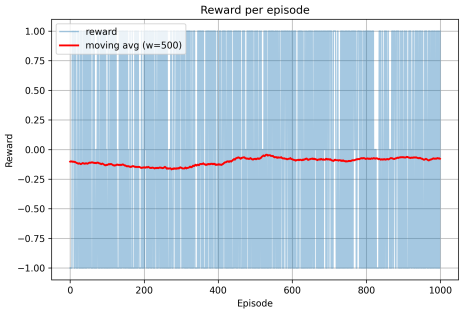
which execution performance is shown in the following image.

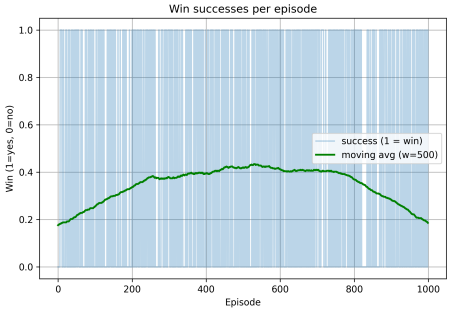


Then we perturbed the parameters to check for improvements on the performance metrics during training, slightly increasing number of episodes and varying the learning rate α and the minibatch size, and also the exploration rate ϵ.
The results of executions are listed below, and then, the same way as with the tabular approach, we compared them to find the best performance setting of the model, which eventually denoted how, in the given configurations, in this game, 2000 episodes with a minor use of the replay buffer outperformed a massively more episode trained network.

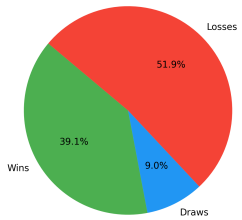
alpha=0.01, epsilon:0.6 - 0.01, epsilon_decay=0.99, gamma=0.9, episodes=1000, buffer_size=1024, minibatch_size=32



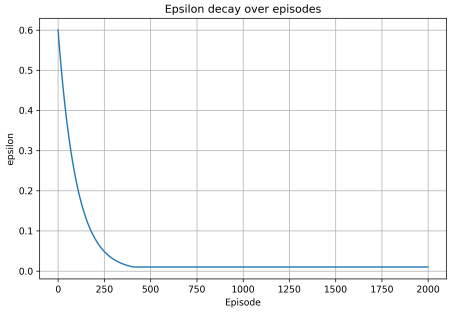


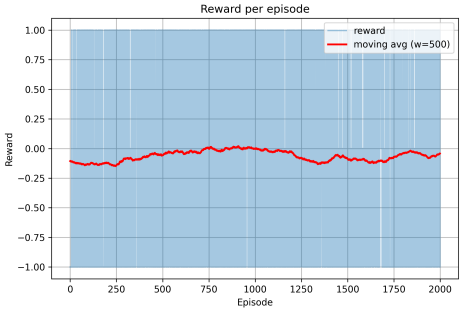


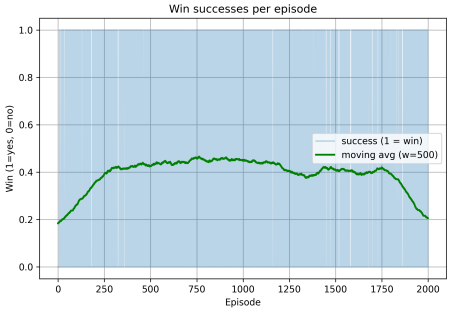
Outcome percentages: wins 391, losses 519, draws 90



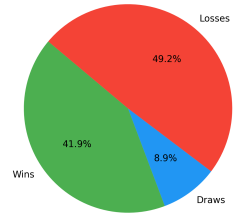
alpha=0.01, epsilon:0.6 - 0.01, epsilon_decay=0.99, gamma=0.9, episodes=2000, buffer_size=1024, minibatch_size=16



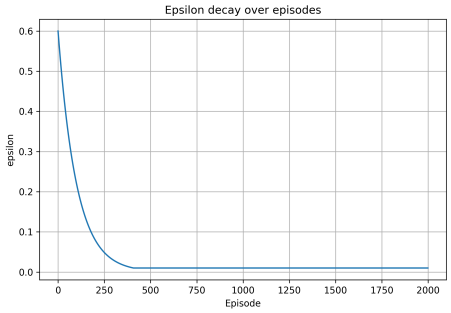


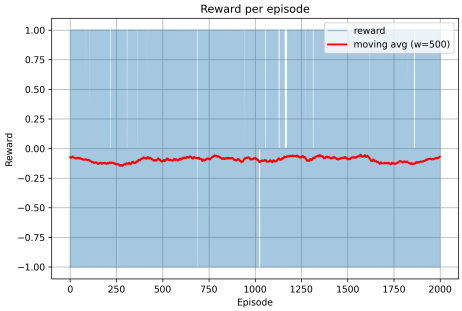


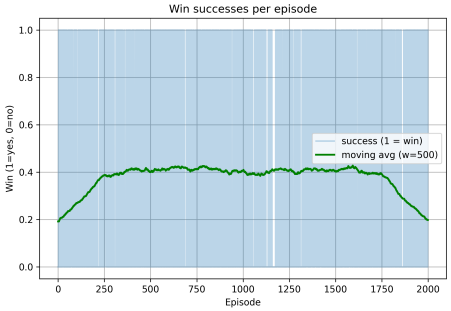
Outcome percentages: wins 837, losses 984, draws 179



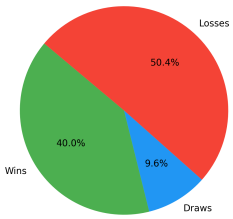
alpha=0.01, epsilon:0.6 - 0.01, epsilon_decay=0.99, gamma=0.9, episodes=2000, buffer_size=1024, minibatch_size=32



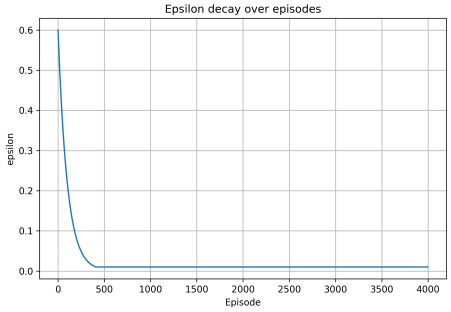


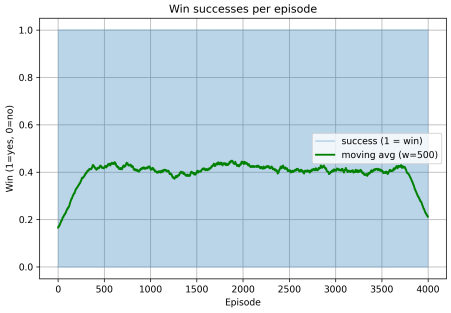


Outcome percentages: wins 800, losses 1009, draws 191

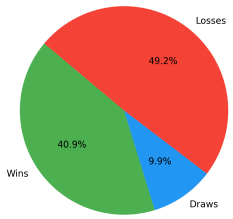


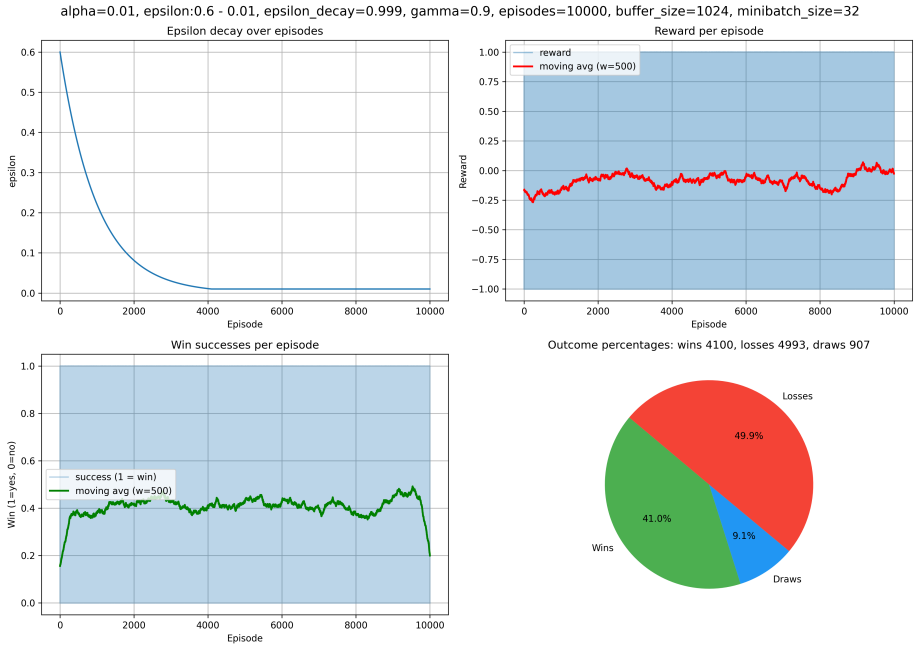
alpha=0.01, epsilon:0.6 - 0.01, epsilon_decay=0.99, gamma=0.9, episodes=4000, buffer_size=1024, minibatch_size=32

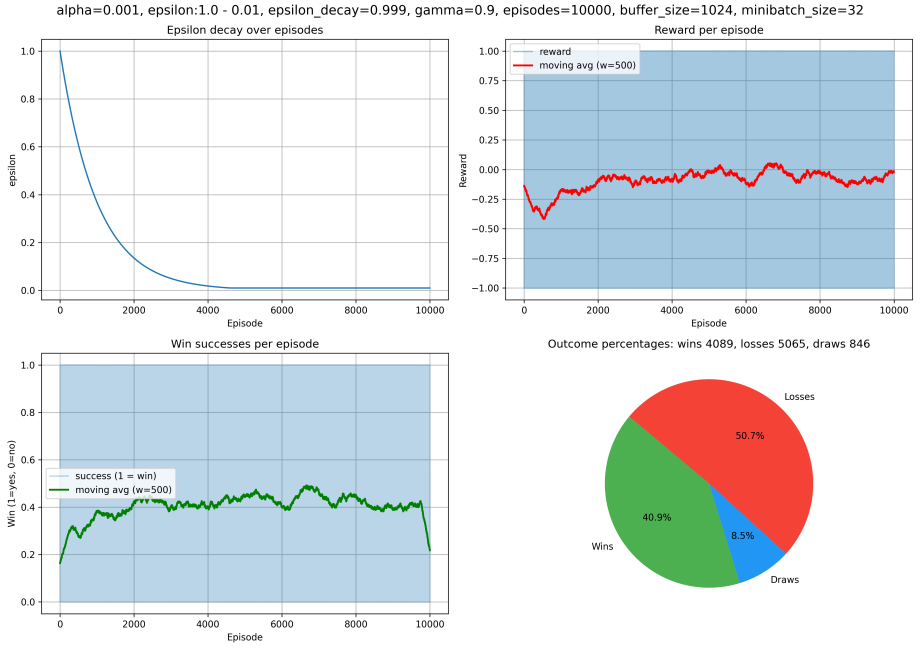




Outcome percentages: wins 1635, losses 1970, draws 395







The results is not much surprising since Blackjack is a card game based on risk. In fact, as stated before, the probability of loss is higher than the

probability of winning. This justifies the results obtained in both tabular and DQN approaches and empirically confirms the percentages of win, loss and draw roughly at 40%, 50% and 10% respectively.