LUISS GUIDO CARLI

THESIS

Blackjack: beating the odds using Reinforcement Learning

Author:
Tommaso AGUDIO

Supervisor:
Alessio MARTINO

A thesis submitted in fulfillment of the requirements for the Bachelor Degree in Management and Computer Science

May 31, 2024

Declaration of Authorship

I, Tommaso AGUDIO, declare that this thesis titled, "Blackjack: beating the odds using Reinforcement Learning" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Tommaso Agudio

Date: 29/05/2024

LUISS GUIDO CARLI

Abstract

Management and Computer Science

Blackjack: beating the odds using Reinforcement Learning

by Tommaso AGUDIO

This thesis studies the application of Reinforcement Learning techniques and their application towards the game of blackjack. The main purpose of the thesis is to see if there is the possibility of overcoming the house advantage, and make one of the fairest casino game also profitable. The study analyzes various algorithms such as Proximal Policy Optimization (PPO), Deep Q-Networks (DQN), and Quantile Regression Deep Q-Networks (QR-DQN), and compares them both in terms of running times and ability to play the game (i.e., number of victories against the dealer) after being trained and optimized with the use of various python libraries. Computational results show that it is indeed possible to beat the odds with the use of Reinforcement Learning (QR-DQN model in particular), but in a simplified version of blackjack that is almost never adopted in real casinos.

Acknowledgements

I would like to express my gratitude to my supervisor, Alessio Martino, for his support and guidance throughout this research project. His expertise in artificial intelligence and computer science really enriched this experience.

I am also thankful to the faculty members of the Management and Computer Science course at Luiss Guido Carli University. Their insightful comments and lessons have significantly enhanced my academic and personal growth.

A special thanks goes to my peers, especially Alessandro Vittorio Natoli and Matteo Carucci, for their help and collaboration throughout these years. Our discussions and efforts have been a crucial part of this academic journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ability of learning machines to be able to learn from experience and improve their performance over time is a topic that fascinated researchers for a long time [6]. Interestingly, games have been a popular testbed for machine learning algorithms and, under a somewhat broader perspective, we witness the same scenario today. In fact, it is not unusual to test not only the performance of learning algorithms, but also the computing power of modern architectures (notably, CPUs and GPUs) by measuring their performance in seamlessly running resource-demanding videogames.

The focus of this thesis is not on the performance of the hardware, but rather on the performance of the software. In particular, we will focus on the ability of several reinforcement learning-based models to learn how to play blackjack. As anticipated, games have been a popular testbed for machine learning algorithms for a long time, and this trend emerged from the 60's: Arthur Lee Samuel, the father of the modern term *"machine learning"* and recipient of the prestigious IEEE Computer Pioneer Award, in 1959, developed an artificial intelligence-based program that learned to play checkers [10].

And yet, we had to wait until the 90's and the early 2000's for more notable results in the field of game-playing artificial intelligence and machine learning, also thanks to the increasing computational power of modern computers. In 1997, IBM's Deep Blue defeated the reigning world chess champion, Garry Kasparov, in a six-game match. In 2011, IBM's Watson defeated the two best *Jeopardy!* players of all time, Ken Jennings and Brad Rutter. In 2016, Google's AlphaGo defeated the world champion of Go, Lee Sedol.

In the remaining part of the following introductory chapter, we will familiarize the reader with the rules of the blackjack and with the concept of reinforcement learning, namely, the core learning paradigm that will be adopted in training and testing the machine learning models.

1.1 An Introduction to the Blackjack Game

First of all, let us introduce the reader to the general rules and the goal of the blackjack game.

The goal of the game is to score 21 points or get as close as possible to it without going over.

During the game, the player will be playing against the dealer or against the dealer and other players. For the purpose of this project, we will consider only the first case: the player against the dealer. In general the player wins if:

- The player's total sum of points is higher than the dealer's total sum of points, or

- The dealer goes over 21 or "busts" (in the game jargon).

The player loses if:

- The player's total is less than the dealer's total.
- The player goes over 21 or "busts".

The player pushes if:

- The player's total is the same as the dealer's total.
- Both the player and the dealer have a blackjack.

In the game jargon, "push" can be seen as a tie.
The game starts with the player and the dealer receiving two cards each.
The player's cards are dealt face up while the dealer has one card face up and one face down.
The player can then choose to "hit" or "stand":

- If the player chooses to hit, he will receive another card.
- If the player chooses to stand he will not receive any other card and the dealer's turn begin.

The dealer will then reveal his face-down card and hit until his total is 17 or higher.
In general, there are other rules, like the player can "double down" or "split" but for the purpose of this project we will not consider them and just use a simplified version of the game. And it is also right to specify that we will be analyzing a situation without bet, meaning that the player plays to win round and not to increase his capital.
The total number of points for both the player and the dealer is given by the sum of the values of their respective cards:

- Cards from 2 to 10 are worth their face value.
- Jacks, Queens and Kings are worth 10.
- Aces are worth 1 or 11, whichever is preferable.

Note that the value of the ace can be 11 in case the current sum of the cards is less than 11, otherwise the player will bust.
It is crucial to note that blackjack is one of the few casino games in which the player can actually use a strategy to increase his chances of winning. This because while it is true that the first cards are based on luck the subsequent actions rely on skill.
One of the most common strategies in blackjack is the counting of cards. But we will not consider it, since it is mostly used to increase the player's capital and not to increase the win rate over time.
Instead, what we will be trying to do is to use reinforcement learning in order to find out if, in the long run, there is a way to beat the odds.

1.2 Reinforcement Learning

The seminal book by Sutton and Barto [11] gives a really good general understanding of what reinforcement learning is, and can be a good starting point for every beginner. Reinforcement learning is a type of machine learning technique that does not fall either in the supervised or unsupervised category. It enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences.
Before going on with the explanation we introduce some basic definition to facilitate the reader to what follows:

- Timestep: a single step or unit of time in the environment
- State space: comprehends all possible states in which the environment can exist.
- Action: the set of all possible moves or decisions the agent can make in a given state
- Reward: the feedback signal given to the agent after it takes an action in a particular state

An agent will interact with the environment, and at each timestep τ, it will retrieve a state S_τ based on a state space S. Then it will take an action A_τ based on an action space A, which will then lead to a reward R_τ based on a reward function R. In general, the goal of the agent is to maximize the expected cumulative reward, which is calculated with this formula:

$$R_\tau = \sum_{k=0}^{\infty} \gamma^k R_{\tau+k+1} \tag{1.1}$$

In this equation γ is the discount factor, which is a value between 0 and 1. It is used to give more importance to the immediate reward rather than the future ones.
In general, it is safe to assume that an agent aims to maximize the reward in the long run rather than in the short run. Therefore, we can assume that in a blackjack profit scenario, the profits will eventually increase after a large number of hands.

1.2.1 Policy

Policy is a core component of Reinforcement Learning, it reflects the strategy adopted by the agent in order to take the best action at a given state and time, and is often represented with the Greek symbol pi: $\pi(s)$. Policies can either be deterministic, where the policy indicates a specific action to take at a given state and time, or stochastic, where the policy provides probabilities for choosing each possible action. As we were saying before, the objective of an agent is to maximize, in the long term, the reward. Since the performance of the policy and the reward are strictly correlated, we can say that the objective of the agent is to find an optimal policy, thus increasing the long term reward.

1.2.2 Exploration vs. Exploitation

The Exploration vs. Exploitation dilemma is one of the biggest challenges in the Reinforcement Learning field:

- Exploration: is the act of trying out new actions in order to discover their rewards. It is crucial because provides to the agent a more clear situation about the whole environment and the potential reward he might gain.
- Exploitation: is the usage of previous agent's knowledge to take an action that is supposed to provide the highest possible reward in a given state and time.

Balancing exploration and exploitation is critical because focusing too much on exploitation can lead the agent to get stuck in a non-optimal policy (local optimum), while excessive exploration can prevent the agent from capitalizing on its knowledge to accumulate rewards efficiently (i.e., akin to a random walk in the search space).

Chapter 2

Algorithms and techniques used

For the creation of the reinforcement learning agent, we used various reinforcement learning algorithms and optimization techniques. We will be analyzing briefly the theory and more in depth the performances of three algorithms:

- Proximal Policy Optimization (PPO)
- Deep Q-Networks (DQN)
- Quantile Regression Deep Q-Networks (QR-DQN)

Since all of these algorithms have a set of hyperparameters that may heavily influence their performance, we used the Bayesian optimization technique [8, 7] in order to find a candidate set of sub-optimal hyperparameters.

It is important to analyze each algorithm individually because, as we will see, each of them employs a different strategy to solve the reinforcement learning problem.

2.1 Proximal Policy Optimization

The authors in [4] have proposed and described Proximal Policy Optimization (PPO) and its possible applications. The idea behind PPO is to improve the policy by making small changes to the parameters at each epoch. It is crucial to avoid big policy changes mainly because it is way more difficult to converge to an optimal policy. It is trivial to think that by making small changes we will eventually converge to the optimal policy. In order to achieve such a result what we do is basically compare the updated policy with the former one, while applying the proximal policy term, which will avoid the future policies to go too far from the former one.

2.1.1 Policy Network Performance

The policy network is a neural network that once given the current state of the environment, based on the various neurons and their weights, will return the optimal action for the agent to take. As we stated before, in PPO, we try to maximize the performance of the policy by making small adjustments to the policy in order to converge to an optimal situation. This behavior directly reflects on the agent's learning curve, making it very smooth and clearly converging. With "learning curve" be simply mean a plot where we represent the reward at a given epoch, and it is trivial to see that a higher reward means a higher "skill" of the agent. To better understand the behavior of PPO, it is useful to plot its learning curve along with the learning curve of a "less smooth" algorithm such as the others used for this project.

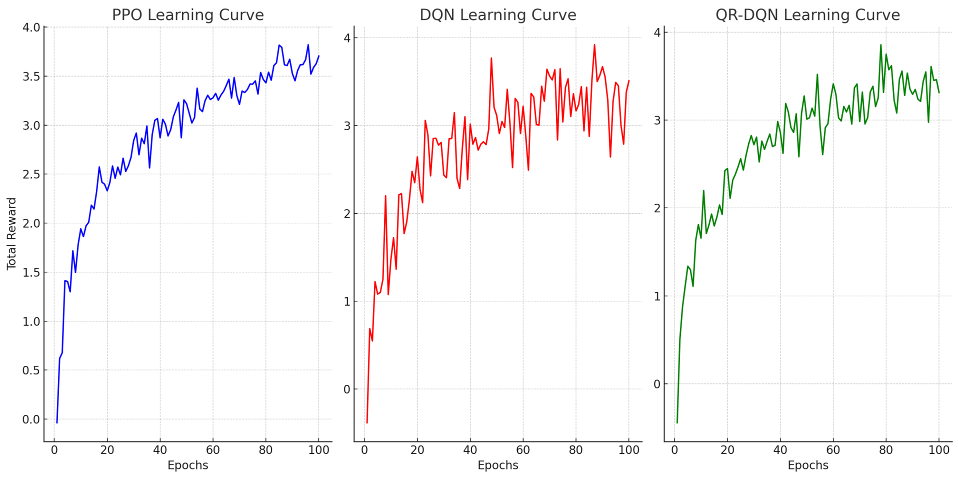

FIGURE 2.1: Learning algorithms learning curves

We can see from Figure 2.1 that the PPO has a smoother and more stable learning curve in comparison to DQN or QR-DQN. For instance, the other algorithms have more fluctuation, reflecting the balance between exploitation and exploration.

2.1.2 Algorithmic details

As we have stated before, the PPO does small adjustments to the policy over time in order to improve it. To achieve such a result it makes use of some specific algorithmic innovations in its objective function. The core of PPO is its objective function, which is designed to minimize the cost of deviation from the old policy while maximizing the expected reward. PPO makes use of a clipping mechanism that is in its objective function and is used in order to avoid overly large policy updates. This is done by clipping the probability ratio of the current policy to the old policy, usually in a range between 0.8 and 1.2. This mechanism is the core of the small policy updates technique, as it ensures that the new policy will not deviate too drastically from the previous iteration.

The PPO's objective function is:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{2.1}$$

where:

- $\hat{\mathbb{E}}_t$ symbolizes the expected value across a finite batch of samples.
- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ represents the ratio of probabilities, indicating how the likelihood of taking action a_t in state s_t under the new policy π_θ compares to the probability under the old policy $\pi_{\theta_{old}}$.
- $\hat{A}_t$ is an estimate of the advantage function at time t, suggesting the relative benefit of taking a specific action compared to the average.
- ϵ is a hyperparameter, typically set to a small value like 0.1 or 0.2, which defines the clipping bounds.
- The function clip$(\cdot)$, depicted in Figure 2.2, constrains the value of $r_t(\theta)$ within the interval $[1-\epsilon, 1+\epsilon]$, ensuring that the policy update does not diverge too drastically from the previous policy.

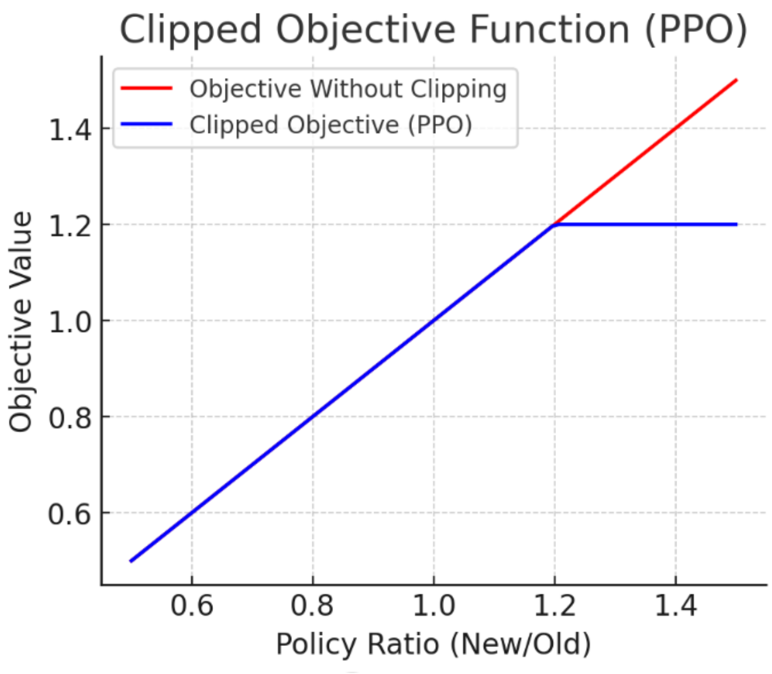

FIGURE 2.2: Pictorial representation of the PPO clipping function.

2.1.3 PPO limitations

Despite all the PPO benefits and performance, it is not free from limitations, which must be taken into consideration when understanding its applicability and performance:

- Hyperparameters sensitivity: like other deep learning algorithms PPO's performance is highly sensitive to the choice of hyperparameters. The most important ones are the clipping range, learning rate and the size of the neural network itself.
- Exploration vs. Exploitation: as we have seen before this problem is common across all the reinforcement learning algorithms, by the way PPO is prone to hinder exploration cause of the mechanism to limit policy updates. This implies that it is harder for the agent to discover new strategies or actions that could lead to higher rewards.
- Computational Resources: PPO requires a significant amount of computational resources, especially in those environments with high-dimensional state or action space.

2.2 DQN

The idea behind Deep Q-Networks (DQN) [9] is to integrate classical Q-learning with deep neural networks. Classical Q-learning uses a Q-table to store the values of the actions and then makes decisions based on that table. On the other hand, in DQN, such tables are replaced with neural networks, to make it possible to achieve better results and handle complex environments. Unlike in PPO, by using small and iterative updates to the network parameters, while carefully controlling the magnitude of policy changes, DQN aims to gradually converge to an optimal policy. Key to this process is the comparison between the updated policy and its predecessor, alongside the application of a term that limits the deviation of future policies from the current policy.

2.2.1 Network Architecture and Performance

The neural networks in DQN are usually referred to as Q-networks, it takes as inputs the current state of the environment and gives as output the Q-values for all possible actions. The focus of the DQN is to maximize the efficiency of the Q-network, and it is done by adjusting the weights of the neural network based on the observed rewards.

2.2.2 Algorithmic Details

DQN updates its Q-values using a loss function that has the objective of minimizing the error between the predicted Q-values and those updated from future reward estimates. A key feature in DQN's algorithmic structure is the use of experience replay and a separate target network, which helps in stabilizing the learning updates. The loss function of DQN is as follows:

$$L(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{2.2}$$

where:

- $L(\theta)$ is the loss function with respect to the neural network parameters θ.
- $\mathbb{E}$ denotes the expectation over the mini-batches of transitions (s, a, r, s') sampled from the replay buffer.
- r is the reward received after taking action a in state s.
- γ is the discount factor that balances immediate and future rewards.
- $Q(s', a'; \theta^-)$ represents the Q-value of the next state s' and all possible actions a', calculated using the parameters of the target network θ^-.
- $Q(s, a; \theta)$ is the predicted Q-value for state s and action a, computed using the current network parameters θ.

2.2.3 Training Process

The DQN training loop involves several key steps:

1. Initialize the replay buffer and the parameters for both the main network and the target network.
2. For each episode, observe the initial state.
3. For each step of the episode:
    - Select an action using an epsilon-greedy policy.
    - Execute the action and observe the reward and the next state.
    - Store the transition (s, a, r, s') in the replay buffer.

- Sample a mini-batch from the replay buffer.
- Compute the target Q-values for each sample in the mini-batch.
- Update the main network by minimizing the loss between the predicted and target Q-values.
- Periodically update the target network parameters.

2.2.4 Limitations of DQN

Despite its successes, DQN has several notable limitations:

- **Sample Efficiency:** DQN often requires a large number of interactions with the environment, which can be computationally expensive and time-consuming.
- **Exploration vs. Exploitation:** DQN can struggle with the balance between exploring new strategies and exploiting known rewards, which may hinder the finding of the most effective policies.
- **Computational Resources:** Implementing DQN requires substantial computational resources, particularly as the complexity of the environment increases.

2.3 QR-DQN

Quantile Regression Deep Q-Networks (QR-DQN) [3, 2] expand upon the foundational concepts of traditional DQN by integrating quantile regression into the learning process. While DQN utilizes a neural network to approximate the mean expected return for each action, QR-DQN estimates the entire distribution of potential returns, providing a richer representation of uncertainty and variability in rewards.

2.3.1 Network Architecture and Performance

In QR-DQN, the architecture is modified to output quantiles of the reward distribution, instead of merely the expected Q-values. This architecture, often referred to as a quantile network, processes inputs from the current environmental state and outputs a set of quantiles that collectively represent the distribution over all possible actions' values. The primary aim in QR-DQN is to enhance the efficiency and accuracy of this quantile network, tuning the neural network's weights based on observed rewards and the distribution they form.

2.3.2 Algorithmic Details

The QR-DQN updates its Q-value estimates using a loss function tailored to minimize the discrepancy between predicted and observed reward distributions. The loss function utilized in QR-DQN is a quantile regression loss, which is defined as:

$$L(\theta) = \sum_{\tau \in T} \sum_{i \in B} \rho_\tau(r + \gamma \hat{Q}(s', \pi(s'); \theta^-) - Q(s, a; \theta)) \tag{2.3}$$

where:

- $L(\theta)$ represents the loss function with respect to the network parameters θ.

- ρ_τ is the quantile loss function for quantile τ, which penalizes the prediction errors asymmetrically depending on whether the prediction undershoots or overshoots the true quantile.
- T is the set of quantiles (e.g., 0.1, 0.5, 0.9) considered in the distribution.
- B refers to the mini-batches of transitions (s, a, r, s') sampled from the replay buffer.
- γ is the discount factor.
- $\hat{Q}(s', \pi(s'); \theta^-)$ indicates the target network's estimated quantile for the next state and action.

2.3.3 Training Process

The training process in QR-DQN involves several distinct steps:

1. Initialize the replay buffer and parameters for both the main quantile network and the target network.
2. Begin each episode by observing the initial state.
3. Throughout each episode step:
    - Select an action using an epsilon-greedy policy adjusted to consider the distributional output.
    - Execute the action, observe the resultant reward and next state.
    - Store the transition in the replay buffer.
    - Sample a mini-batch from the replay buffer.
    - Compute the target quantiles for each sampled transition.
    - Update the main network by minimizing the quantile regression loss between the predicted and target quantiles.
    - Periodically refresh the parameters of the target network to stabilize training.

2.3.4 Limitations of QR-DQN

Despite its advancements over DQN, QR-DQN encounters its own set of challenges:

- **Sample Efficiency:** Like DQN, QR-DQN can require extensive interactions with the environment, which may still be computationally demanding.
- **Complexity in Implementation:** The addition of quantile regression introduces greater complexity in the network architecture and the training process.
- **Resource Demands:** The need to estimate a full distribution rather than a single expected value increases the demand on computational resources, particularly in complex environments where the variability in returns can be significant.

QR-DQN represents a sophisticated evolution in reinforcement learning, aiming to capture a more comprehensive understanding of the potential outcomes and their probabilities, thereby enabling more informed decision-making under uncertainty.

Chapter 3

Experimental Setup and Computational Results

In this chapter we will talk about how we created the environment for the agent, the reasoning behind certain choices, and how we implemented the Reinforcement Learning algorithms discussed in Chapter 2. The objective is to maximize the win rate of the agent against the dealer in the simplest blackjack scenario possible, that is:

- There will be only the Agent against the Dealer, no other players
- The will not be:
  - Splitting
  - Insurance
  - Double
  - Sidebets
- No betting system (we are focusing on maximizing the win rate, not the profit).

3.1 Environment setup

In the creation of the environment where the agent will be trained it is crucial to define what the event and observation space will be. This is because the environment dictates how the agent will interact with the environment and how it sees the state of the game.
For the creation of the environment, and the management of the various settings of it, we will be using the library "Gymnasium" [13]. It is the successor of OpenAI's library named "Gym" [12]. The main purpose of Gymnasium is to provide an easy way to create, manage, and customize new or existing environments.
Creating a new environment in Gymnasium is pretty easy: we just make use of the predefined class, that can be created with a code snippet like this:

```python
import gymnasium as gym
from gymnasium import spaces

class CustomEnv(gym.Env):
    def __init__(self):
        super(CustomEnv, self).__init__()
        # Define action and observation space
        # They must be gymnasium.spaces objects
        # Example: self.action_space = spaces.Discrete(2)
        # Example: self.observation_space = spaces.Box(low=0, high=1, shape=(3,), dtype=np.float32)

        self.action_space = spaces.Discrete(2)  # Example: Two discrete actions
        self.observation_space = spaces.Box(low=0, high=1, shape=(3,), dtype=np.float32)  # Example: Continuous observation space

    def reset(self):
        # Reset the state of the environment to an initial state
        # Return the initial observation

        initial_observation = self.observation_space.sample()  # Example: Random initial observation
        return initial_observation

    def step(self, action):
        # Execute one time step within the environment
        # This method must return four values:
        # - observation: The observation of the current state
        # - reward: The reward obtained after executing the action
        # - done: Whether the episode has ended
        # - info: Additional information, typically for debugging

        observation = self.observation_space.sample()  # Example: Random next observation
        reward = 0  # Example: Reward placeholder
        done = False  # Example: Episode end condition placeholder
        info = {}  # Example: Additional information placeholder

        return observation, reward, done, info

    def render(self, mode='human'):
        # Render the environment to the screen or other output
        pass  # Example: Rendering placeholder
```

We will not paste the code snippet of the blackjack environment because it would be too long to show. However, let us delve deeper into what this code snippet does and how we used each method of the class in order to build our environment.

- `__init__`: Initializes the environment, defining the action and observation spaces. Here, the action space is set to Discrete(2), which means there are two possible discrete actions. The observation space is set to a continuous space with values between 0 and 1. In our case in the init method we created four main variables, that are:
  - initial-deck: it contains values from 1 to 10 representing the cards, the 10 is repeated 4 times in order to track the figures and the card "10". Also we multiplied this list of values by four, in order to simulate the presence of four decks.
  - action-space: this represent the size of the action space, in our case we opted for a discrete(2), which means that there are two possible actions, 1 (hit) or 2 (stand).
  - Observation-space: in our case we created an array of size 23 that contains values from 1 to 11. The first eleven elements of the array refers to the player's hand, the other eleven to the dealer's and the last one is used to check for "usable ace".
- reset: Resets the environment to its initial state and returns the initial observation. In this template, the initial observation is a random sample from the observation space. In our code the reset function is pretty basic. The first thing that it does is, using an if statement, checking if the deck has less then 15 cards, in case that it is true the deck is recreated. Then it draws the initial hands to both the dealer and the player, with the function "Draw hand".
- step: Executes one time step within the environment. It takes an action as input and returns the next observation, reward, done flag, and additional info. The provided template uses placeholders for these values. This is a crucial part of the environment, as this dictate the way how the whole game will work. Our code start with an if statement, checking whether the player decided to hit or stand. In case he decided to hit the code will then check if the player has busted, in case he has the hand is counted as a loss, otherwise the player can decide again whether to hit or stand. In case he decides to stand then the dealer's turn begins, simply the code will draw random cards until the dealer either busts or his hand's value reached 16. In this section of the environment also the reward mechanism is implemented. In case that the player busts he will get a -1 as reward, in case that he wins he receive +1, if he loses -1 and if it is a draw 0.
- render: Placeholder method for rendering the environment. This can be expanded to include visualization of the environment's current state. In our code we opted for a "human" rendering, that just gives us a look at the current state of the game, showing the dealer's and player's hand at each moment of the game.

In our code also another version of the environment can be found, named "deterministic-env". As the name suggests, it is a deterministic blackjack situation, meaning that all the randomness of the game is removed and the same deck will be used over and over again. Clearly, in a real life scenario, this would not make any sense but, in our case, it was proven useful in order to check if the agent was properly learning the game, finding strategies and recognizing patterns in a controlled (i.e., deterministic) environment. The usage of this deterministic environment will not be taken into consideration for the upcoming models evaluations, but it played a crucial role in the testing phase.

3.2 Models creation and optimization

As anticipated in Chapter 2, we will be using three models in order to try to beat the odds: PPO, DQN and QR-DQN. All of these models have been not implemented from scratch, instead we leveraged a library named StableBaselines [5]. This library makes it very easy to create reinforcement learning models, it supports both very common models and not so common models (such as QR-DQN). Of course you get the best out of the library by fine tuning the models, instead of using the default parameters. In order to find the best hyperparameters we used another library, named Optuna [1]. Optuna makes it fairly straightforward to find a suitable set of hyperparameters by using various optimization algorithms. The main strengths of Optuna are:

- Easy integration: It is designed to work seamlessly with popular machine learning libraries.
- Visualization: It includes utilities for visualizing the optimization process, which can help in understanding the hyperparameter effects and the convergence of the search.
- Parallelization: Optuna supports easy parallelization of trials, enabling the distribution of the hyperparameter search across multiple CPUs or even different machines.

Before getting straight into the creation of the models we will give an overview of the entire pipeline towards model creation. First of all, we had to decide the number of timesteps. We opted to train each model two times: at first, with 500k timesteps and then with 1 million timestamps. This will let us better understand how the number of timesteps influence the performance of the agent. Then we had to decide how to keep track of the progress, like: what do we want to measure? The number of win? Of losses? The number of times that the agent hits or stands? We kept track of all of these information, and all of them will then be used to make various graphs that makes it easy to understand the behaviour of the agent. Now the next thing that we had to decide was: how do we tackle this? To answer this question we will now get into the details of each step for the creation of the models.
First of all we imported the libraries and created three different templates of code, one for PPO, one for DQN and one for QR-DQN. We will not see each single template, instead we will give a look to the code snippet of the DQN template step by step.

```
def evaluate_agent(model, env, num_games=1000):
    wins = 0
    win_rates = []
    num_games_list = []  # List to store the number of games after each
        logging interval

    for i in range(num_games):
        obs = env.reset()
        done = False
        while not done:
            action, _ = model.predict(obs, deterministic=True)
            obs, reward, done, _ = env.step(action)
            if done and reward == 1:
                wins += 1
        if (i + 1) % 100 == 0:  # Log win rate every 100 games
            win_rates.append(wins / (i + 1))
            num_games_list.append(i + 1)  # Append the number of games
                played so far

    # Create a DataFrame with both win rates and number of games
    win_rate_df = pd.DataFrame({'WinRate': win_rates, 'NumGames':
        num_games_list})
    win_rate_df.to_csv('DQN500k_win_rate_over_time.csv', index=False)

    return wins / num_games
```

We first created the function "Evaluate agent". This function will let use retrieve the win rate of the trained model by simulating 1000 games, and the results will then be converted into a Pandas DataFrame, in order to make it easy the manipulation and visualization of the results. Note that to improve readability and have a dataset as clean as possible we logged the win rate each 100 games.

```
env = DummyVecEnv([lambda: SimpleBlackjackEnv()])
model = DQN(
    "MlpPolicy",
    env,
    learning_rate=1e-3,
    buffer_size=10000,
    learning_starts=1000,
    batch_size=32,
    tau=1.0,
    gamma=0.99,
    train_freq=4,
    gradient_steps=1,
    target_update_interval=1000,
    exploration_fraction=0.1,
    exploration_final_eps=0.02,
    max_grad_norm=10,
    tensorboard_log="./blackjack_dqn_tensorboard/",
    verbose=1
)
model.learn(total_timesteps = 50000)
```

In this part of the code we create and train our model. The first thing that we have done is to import the environment that we have created before with the "DummyVecEnv" function. Right after that we declared the model that we want to train with the hyperparameters, since we do not want to use the default ones. The hyperparameters were decided with the use of another code snippet, that runs the Optuna library in order to retrieve the best hyperparameters. We then start the learning of the model with "model.learn", specifying the number of timesteps.

```
def simulate_blackjack_games(env, model, num_games=10000):
    action_frequencies = {}
    rewards = []
    results = []
    for game in range(num_games):
        obs = env.reset()
        done = False
        total_reward = 0
        player_actions = []
        player_hand_sums = []
        while not done:
            action, _ = model.predict(obs)
            player_actions.append('Hit' if action == 1 else 'Stick')
            # Define state key
            player_hand = obs[:11][obs[:11] != 0]
            dealer_visible_card = env.dealer[0]
            state_key = (tuple(player_hand), dealer_visible_card)
            # Record action frequencies
            if state_key not in action_frequencies:
                action_frequencies[state_key] = {'Hit': 0, 'Stick': 0}
            action_frequencies[state_key]['Hit' if action == 1 else '
                Stick'] += 1
            obs, reward, done, _ = env.step(action)
            total_reward += reward
            player_hand_sums.append(env.sum_hand(player_hand))
        rewards.append(total_reward)
        player_final_hand = obs[:11][obs[:11] != 0]
        dealer_final_hand = obs[11:22][obs[11:22] != 0]
        game_results = {
            'Game': game + 1,
            'PlayerFinalHandSum': env.sum_hand(player_final_hand),
            'DealerFinalHandSum': env.sum_hand(dealer_final_hand),
            'PlayerNumCards': len(player_final_hand),
            'DealerNumCards': len(dealer_final_hand),
            'DealerVisibleCard': dealer_visible_card,
            'PlayerActions': ' '.join(player_actions),
            'PlayerHandProgression': ' '.join(map(str, player_hand_sums
                )),
            'Outcome': 'Win' if reward > 0 else 'Loss' if reward < 0
                else 'Draw'
        }
        results.append(game_results)
    action_freq_data = []
    for state, actions in action_frequencies.items():
        player_hand, dealer_card = state
        action_freq_data.append({'PlayerHand': ' '.join(map(str,
            player_hand)),
                                 'DealerVisibleCard': dealer_card,
                                 'Hit': actions['Hit'],
                                 'Stick': actions['Stick']})
```

In this last snippet of code for the creation of the model we have just simulated 10k games, but this time we kept track of all those information that we thought might be useful to better understand the performance of the model. For instance we have stored are the player's actions (hit or stand), the player's and dealer's hands value, the player's and dealer's final hand sum, if the agent has won or lost and the obtained reward for each single game.
This code snippet will change slightly among the various models, the only things

that will change are the hyperparameters of the model and some strings, but the majority of the code will remain unchanged. Let us now talk about the hyperparametrs that we tuned for each single model, note that we will not repeat the hyperparametrs that we have already explained another model:

- PPO:

  learning_rate (2.5×10^{-4})**:** This is the step size used for updating the weights of the neural network. The objective here is to use a value small enough to ensure smooth and stable updates without causing large swings in policy performance, that we have seen before is a crucial point of the PPO.

  n_steps (256): This parameter defines the number of steps to collect in each environment per policy rollout before updating the model. Larger values can lead to more stable policy updates because the updates are based on a broader experience.

  batch_size (64): Batch size controls the number of training examples utilized to perform a single update of the model's weights. Larger batch sizes provide smoother gradients but at the cost of increased memory and processing power. Larger batch sizes can also improve the training time, but once again at the cost of increased processing power.

  n_epochs (10): Refers to the number of complete passes through the entire dataset (collected in n_steps) before the data is discarded. Multiple epochs help in thoroughly utilizing the collected experiences, potentially leading to better learning outcomes by refining the policy incrementally with repeated exposure to the same data.

  gamma (0.99): The discount factor, gamma, affects how future rewards are valued relative to immediate rewards. A high value like 0.99 emphasizes the importance of future rewards, encouraging strategies that may reap benefits over a longer horizon rather than optimizing for immediate gains. This can be crucial in environments where the consequences of actions manifest over extended periods.

  gae_lambda (0.95): This parameter in Generalized Advantage Estimation moderates the trade-off between bias and variance in the advantage estimates, which are used to update the policy.

  clip_range (0.2): In PPO, the policy update is clipped to remain within a specified range to prevent destabilizing large updates. The clipping range that we have set (0.2) means that the policy update ratios (new policy probability divided by old policy probability) are kept within 20% of 1 (i.e., between 0.8 and 1.2). This helps in balancing exploration and exploitation by allowing the policy to improve while avoiding drastic changes based solely on high-variance estimates.

  ent_coef (1×10^{-4})**:** The entropy coefficient promotes policy exploration by adding a term to the objective function that rewards higher entropy, or randomness, in the policy's action distribution. This small value indicates a subtle influence, ensuring that exploration does not override the primary goal of optimizing the policy but still facilitates sufficient randomness to explore new strategies.

- DQN:

MlpPolicy: This defines the policy model used in the DQN, specifying a Multi-Layer Perceptron (MLP) as the architecture for approximating the Q-value functions. MLPs are feedforward neural networks often used for tabular data.

learning_rate (1e-3): The learning rate specifies how much the weights of the neural network are updated during training. A smaller value means slower learning, and a larger value could lead to overshooting the optimal values. The rate of 1e-3 is a typical choice for steady convergence.

buffer_size (10000): This determines the size of the replay buffer. The replay buffer stores transitions collected from the environment, which are tuples of (state, action, reward, next state). A larger buffer can store more experiences, enabling learning from a wider range of past actions.

learning_starts (1000): This parameter defines how many steps of the environment are collected with the initial random policy before learning starts. This helps to populate the replay buffer with enough experiences to begin meaningful learning.

tau (1.0): Tau is used in soft updates of the target network's weights. A value of 1.0 indicates that the target network is hard updated, i.e., directly copied from the local network. In contrast, smaller values closer to zero would indicate softer updates, blending the weights gradually.

train_freq (4): Defines how often the network is trained. Here, the model trains every four steps, which balances between too frequent updates (which can be computationally expensive) and too infrequent updates (which can slow down learning).

gradient_steps (1): This parameter sets how many gradient steps are taken per training step. Each gradient step involves one backward pass of the network to update the weights.

target_update_interval (1000): Determines how frequently the target network's weights are updated with the weights from the model. Frequent updates mean the target network closely tracks the learning model, while less frequent updates can stabilize training.

exploration_fraction (0.1): This fraction of the total number of steps to be taken within which the exploration rate (epsilon) linearly decreases from 1 to the exploration_final_eps.

exploration_final_eps (0.02): The final value of epsilon in the epsilon-greedy policy. Epsilon controls the trade-off between exploration (choosing a random action) and exploitation (choosing actions based on the learned values). A lower final epsilon value emphasizes exploitation.

max_grad_norm (10): This sets a limit on the norm of the gradients. Clipping the gradients at this norm helps in preventing the exploding gradient problem, which can lead to unstable training dynamics.

Note that we didn't got into the details of the QRDQN's hyperparameters since we tuned the same parameters of the DQN, but with different values.

3.3 Training and evaluating the models

Let us now begin to train the models and see in detail the performance of each single model to then compare the models against each other. But before talking about the first model let us create a sort of pattern that we will be following to have a "leveled playing field" for the evaluations of the models. We will take into account:

- Time to train
- Win rate
- Win rate over time
- Player's hand distribution

Those four parameters will be the one that will be compared between the models.

3.4 PPO

Let us start by analyzing the PPO model. Like all the other models, it underwent 2 training stages: first with 500k timesteps and then with 1M timesteps. PPO is the lighter and less computational power demanding model of the three, therefore it will have the fastest training time. We already have looked at the hyperparameters that we have tuned for this model, so let us get right into its performance. Specifically, let us start by analyzing the training time. The PPO took as little as 1 minute to train for the 500k timesteps and 3 minutes for the 1 million timesteps. Both of them were trained on a Macbook with Apple Silicon M2 CPU and 8GB of RAM. As we will see later on, the PPO is the only model that can be trained on a normal machine with fast training time. Let us now introduce the win rate of the model, in the following graph we can see the number of win, loss and draw of the agent:

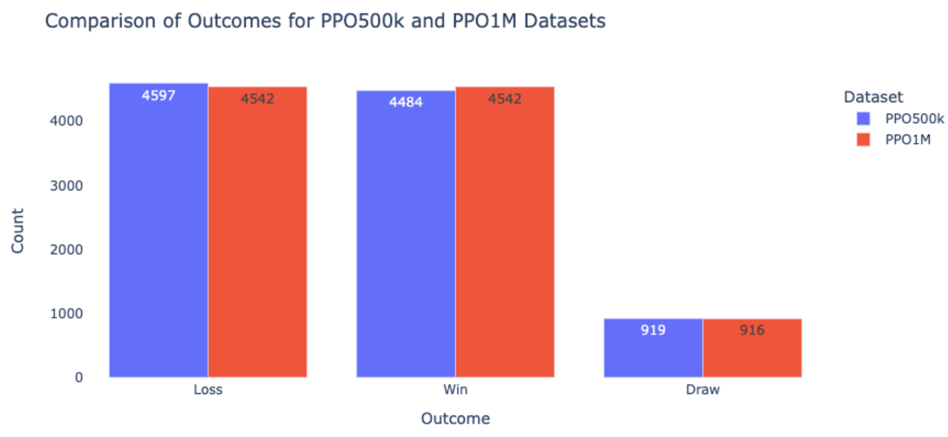

FIGURE 3.1: PPO win rate

As we can see from Figure 3.1, the performance are not the best, for instance the model was able to achieve only a win rate of 44.84% (with 500k timesteps) and 45.52% (with 1 million timestamps). But if we consider a draw as win (since we are not losing money, theoretically) the model is performing even better. We want to recall that the average win rate in a real world blackjack scenario is around 42.2%, with a 6.5% of draws and 51.2% of loss. Another interesting statistic to look at is the distribution of the hands value, for both the player and the dealer. Note that the dealer hands distribution will be the same across all the models, since it changes just slightly and is not dictated from strategy but from luck. By plotting them we got the following graph:

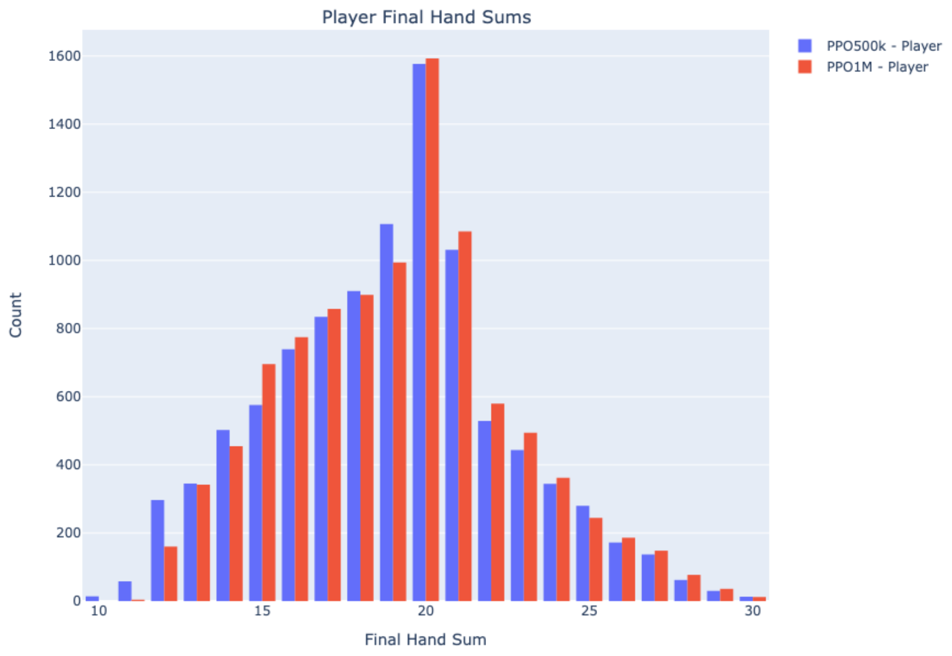

FIGURE 3.2: PPO hands distributions

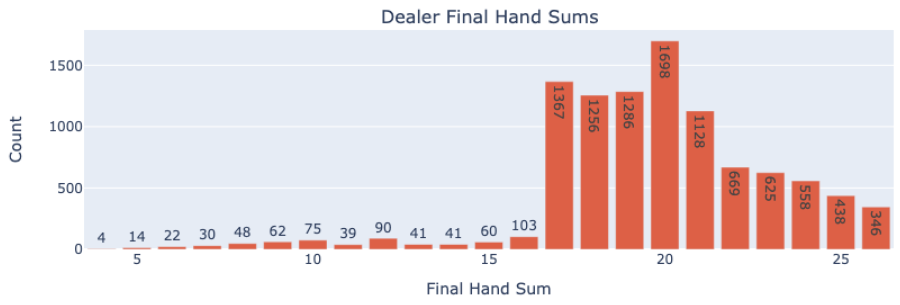

FIGURE 3.3: Dealer's hands distributions

By matching Figures 3.3 and 3.2 we can see the agent's hands are almost normally distributed, slightly right skewed. While, on the other hand, the dealer's hands are completely left skewed: this is due to the rules of the game, that constraint the dealer to ask for card until 16 or more. An interesting thing that we can see is that the dealer got around 700 more blackjacks compared to both models, and this applies to all the high-values of the hands. This is still probably due to the rules of the game. Talking about the win rate over time we can see it in this plot:

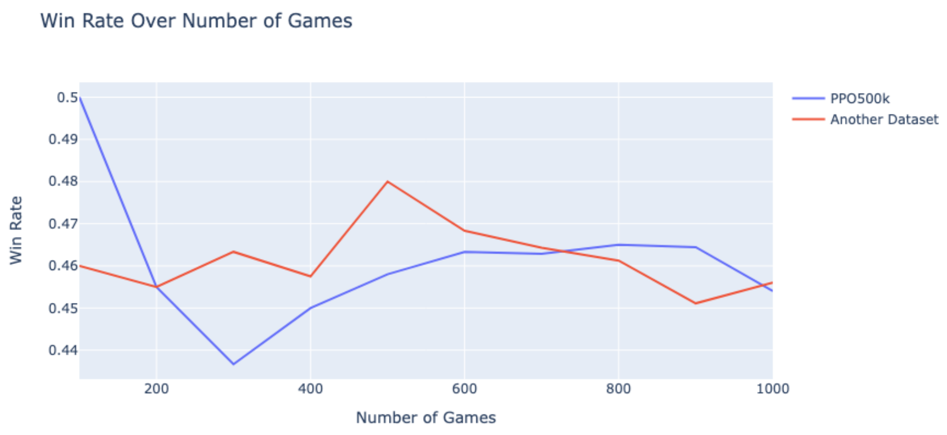

FIGURE 3.4: PPO win rate over time

Figure 3.4 shows the win rate each 100 games until 1000. We can see the in the first 100 the model with 500k timesteps had a 50% of win rate, which is pretty high, but this hasn't lasted long, since it dropped down 45.5% in the 200s to then achieve a final 45.5%. The high starting win rate might due to luck or band dealer's hands, but in the long run the strategy developed by the model showed that it is not the best. On the other hand the 1 million model had a lower starting win rate, that got high in the middle to then finish just a little above the 500k model, with a 45.6%
In general we can say that the PPO model achieved good results, especially if we consider the fast training time and the fact that it can be trained on any computer. It got results that are above the win rate of real life scenarios, meaning that it uses strategies similar to real player's one or even better.

3.5 DQN

Let us now analyze the DQN model, like for the PPO we have already seen the hyperparameters and also talked about the theory behind it, so we can start analyzing its performance and, as for the PPO, we trained it with 500k and 1 million timesteps. We will start again from the training time. This model is way more computational intense and will for sure lead to a normal computer begin fully utilized, from the RAM to the CPU/GPU power. Also, the training time would be so long on a normal computer. For instance, we tried to train it on the same Macbook already used for training the PPO and it lead to a waste of 2 hours, since in 2 hours it trained only for 50k timesteps to then stop due to the high CPU temperatures. For this reason, we used a workstation with dedicated GPUs and high RAM[1]. With this workstation, the running time were more feasible ranging from 20 to 40 minutes.

Let us see the win rate of the model:

[1]Intel(R) Core(TM) i9-9940X CPU @ 3.30GHz, 128 GB RAM, GPU NVIDIA GeForce RTX 2080 Ti.

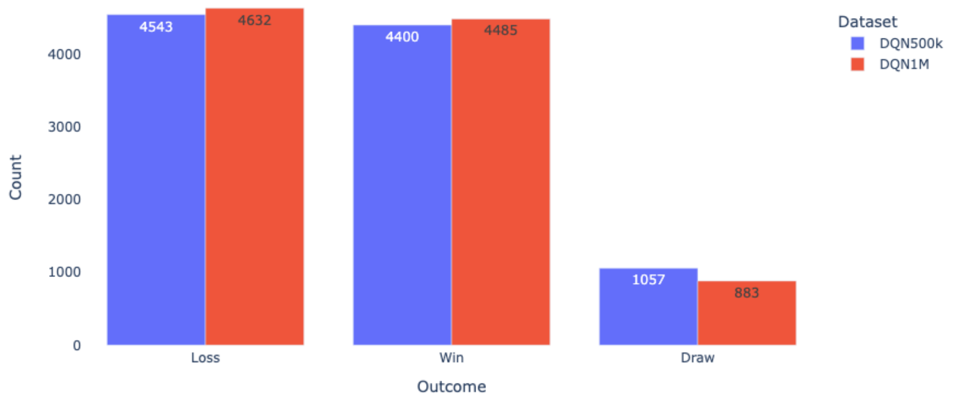

FIGURE 3.5: DQN win rate

As we can see from Figure 3.5 we have win rates that are not that high, and the amount of loss is still higher then the amount of wins. For instance we can see that with 500k timesteps we got a 44% of win rate, while with 1 million we got 44.85%. But if we consider the draw as a win, we can once again see those results as acceptable.

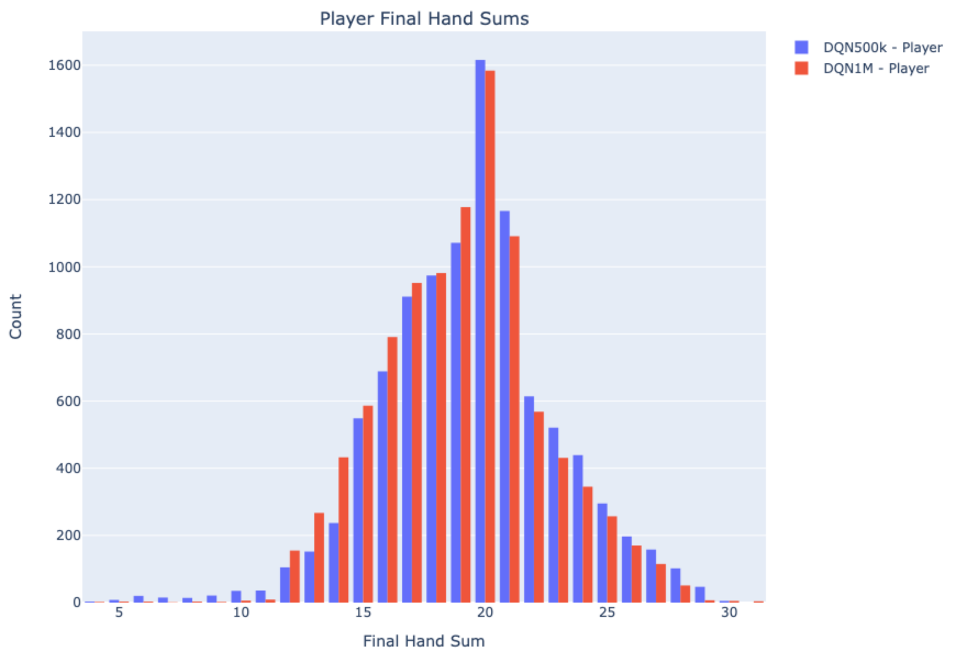

FIGURE 3.6: DQN hands distribution

As we can see from Figure 3.6, here the models tends to not bust, and has achieved a lot of 20s. If we consider again the previous dealer's hand distribution we can see that he got roughly 400 more blackjacks compared to the model. This can be once again attributed to the rules of the game, that make the dealer more prone to high hand values.

The win rate over time is as follows:

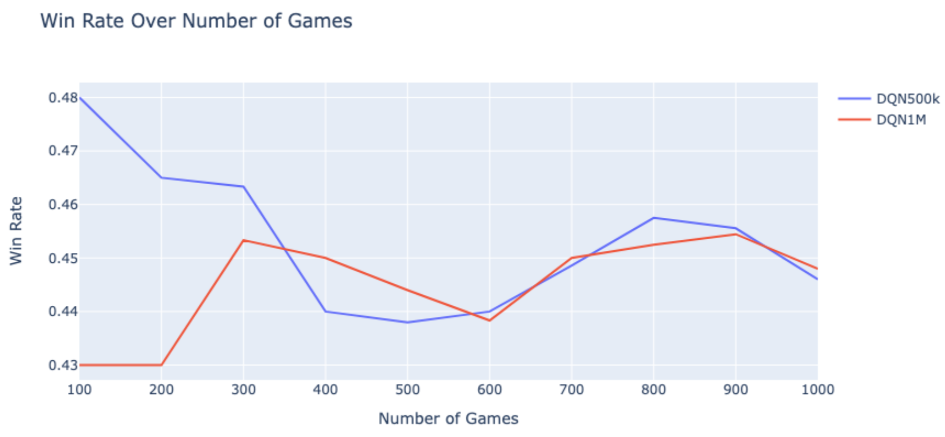

FIGURE 3.7: DQN win rate over time

As we can see from Figure 3.7, the 500k model started with a pretty high win rate (48%), to then drop very low and climb up to finish at 44.6%. On the other hand the 1M model has done the exact opposite, starting really low to end slightly above the 500k model, with a 44.8% of win rate. This difference in start is most probably due to randomness, as in the long run we see that both lines tend to overlap, meaning that the model is prone to perform the same with 500k or 1 million timesteps. This also mean that most probably both models adopt the same strategy of playing the game.
In conclusion we cannot really say that DQN is a good choice for this purpose. This due to several factors, the main one begin that it takes long to train and required a well-equipped workstation. Also it performed almost like a normal strategy, making the whole effort of creation and training of such a model not worth.

3.6 QR-DQN

Like for the previous models, we trained as well the QR-DQN with 500k and 1 million timesteps. Regarding the training time we can directly say that this was the worst performing model among the three. Like the DQN, it was impossible to train it on a normal computer, and using a dedicated computer took us nearly an hours to train. But was this effort worth it?

Let us see how the win rate went for this model:

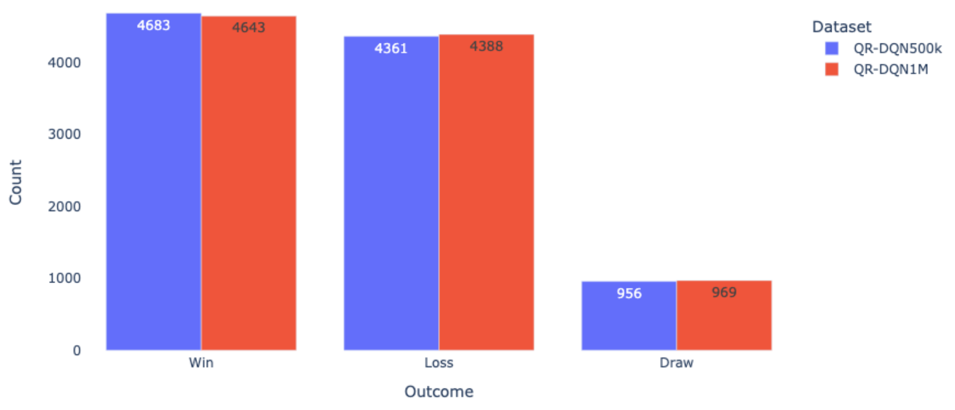

FIGURE 3.8: QR-DQN win rate over time

The first thing that we can notice from Figure 3.8 is that both models achieved more wins than losses, meaning that they were profitable if we played money. The win rates are similar, the 500k model got 46.83% and the 1 million got 46.43%. The models performed really well, going well above the average win rate, but the most important fact is still that they had more wins than loss.

For what it regards the hands distribution, is as follows:

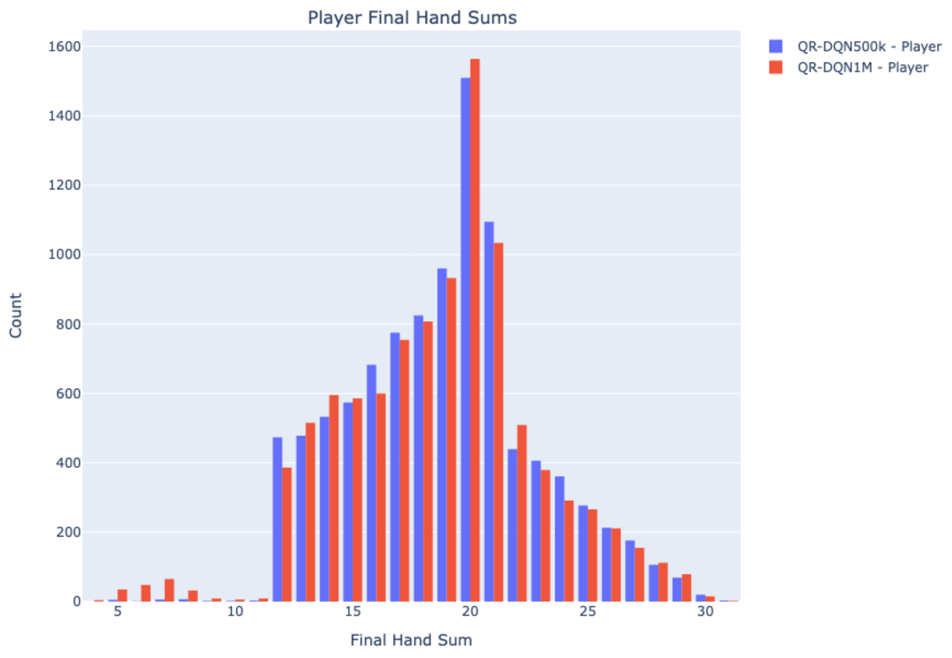

FIGURE 3.9: QR-DQN hands distribution

From this distribution displayed in Figure 3.9 we can see that the model got a lot of 20s and also the number of blackjack increased. It looks like that the number of busts decreased a little bit, showing a bit more of a strategy. If we take into consideration the previous dealer's hands distribution graph we can see that the dealer got just 100 more blackjacks compared to the model, reflecting the higher win rate of the model.

Talking about the win rate over time, we shown them in Figure 3.10:

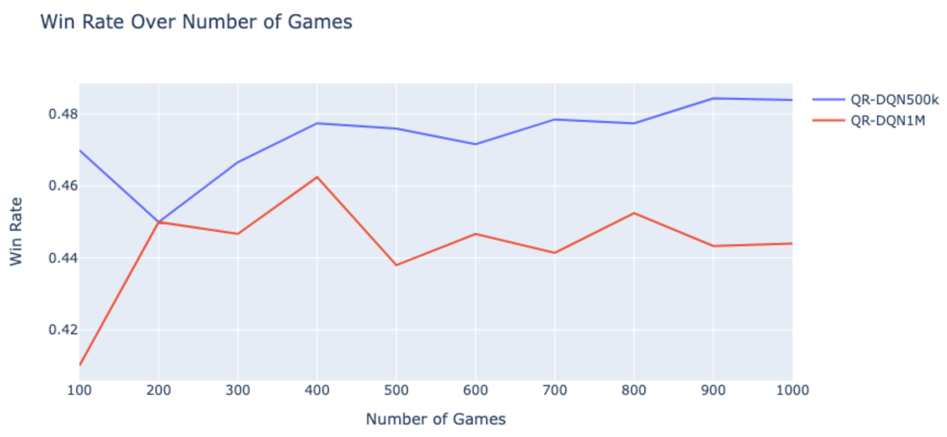

FIGURE 3.10: QR-DQN win rate over time

Specifically, we can see a pretty different behaviour between the models. The one with 500k timesteps started high to then go higher, while on the other hand the 1 million model started low to then try to catch up. To get a bit more in details the 1 million model started with a 41% of win rate, to increase until the end where it got a not so good 44.4% of win rate. On the other hand the 500k model started high with a 47% of win rate ending with a stunning 48% win rate.
In conclusion the QR-DQN performed really well, but it still requires really high computational power. The models performed almost the same, just in the simulation regarding the win rate over time they differed a lot, this might be due to the stochastic nature of blackjack. The 500k model performed better in general, meaning that the QR-DQN tends to find a good strategy already in the early stages.

Chapter 4

Discussion and Conclusions

To conclude this thesis, let us now compare all the models together, to then get some conclusions regarding the models, blackjack and gambling as a whole. We will follow the same procedure, starting from the training time. To simplify the visualization of the running times and facilitate discussion, we have summarized those in a table. Note that the running times refer to the same workstation with dedicated GPUs, the PPO training time change of just some seconds so we rounded them.

	500k	1 Million
PPO	1	3
DQN	30	40
QR-DQN	60	90

TABLE 4.1: Training time (in minutes)

As we can see from Table 4.1, the best model in terms of training time is undoubtedly the PPO, requiring as low as 1 minute to train. As we have seen, and will see, this low training time comes to the price of lower win rate compared to the QR-DQN. But this is still a crucial point of a model, because if we are just looking to have fun, or we want to try a lot of different hyperparameters having a fast training model will make this process way easier.

Let us now talk about the distribution of the hands. Before plotting the distribution we have to specify that we had to zoom the graph and get a focus on the higher values of the graph. This mainly due to the fact that the graph is too big and confusing due to the number of models begin plotted. But, if we zoom in into the right-hand of the graph, it becomes well understandable and useful.

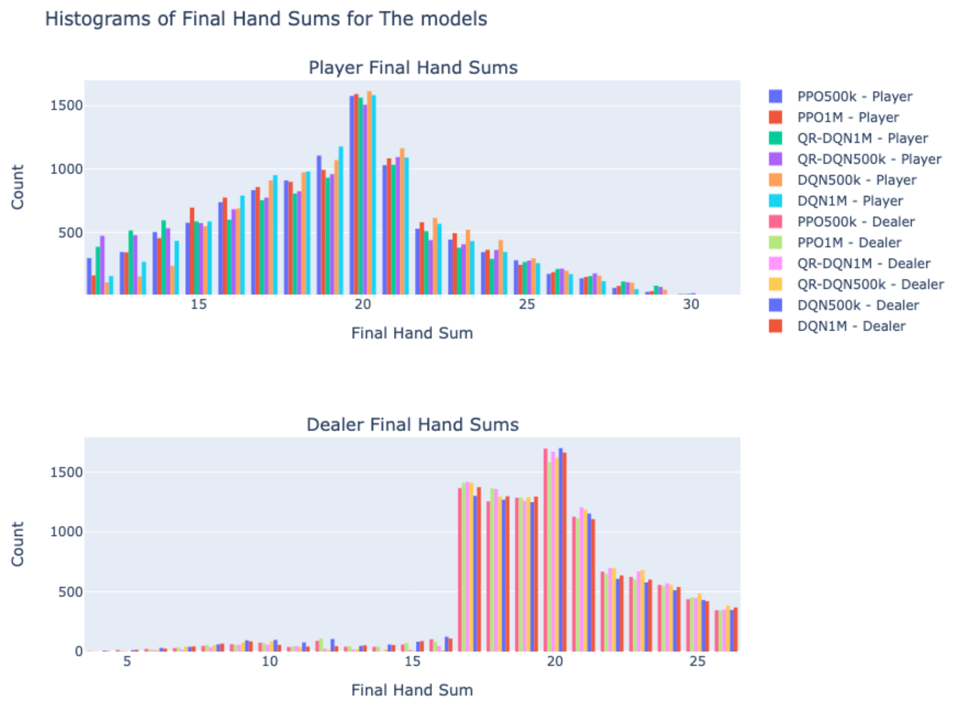

FIGURE 4.1: Models' hands distribution

First of all, as we can see in Figure 4.1, we have also plotted the dealer's hands distribution just to show that they are practically identical among different runs. In general all the models performed similarly, but let's see more in detail. We will analyze the number of 19s, 20s and blackjack that the models got, since those values are the highest value that a normal player looks for.

	19	20	BlackJack
PPO500k	1107	1577	1031
PPO1M	994	1593	1085
DQN500k	1071	1616	1166
DQN1M	1178	1584	1091
QR-DQN500k	960	1509	1095
QR-DQN1M	932	1564	1034

TABLE 4.2: Number of 19s, 20s, blackjacks

As we can see from Table 4.2, the highest numbers of blackjacks has been achieved by the DQN with 500k timesteps, that also got the highest number of 20s. But as we will see later on this hasn't led to higher win rates, probably due to the fact that the model was more risk seeker, asking for card even in those situations where it was not necessary. On the other hand, the lowest number of blackjacks has been achieved by the PPO with 500k timesteps, reflecting the not so high win rate of the model.
This table and the short analysis that we have done will help us to get a better overview of the scenario.
Now let us talk about the most interesting part, the win rates of the models and the win rates over time of the models. Starting from the win rates, we can see the plot:

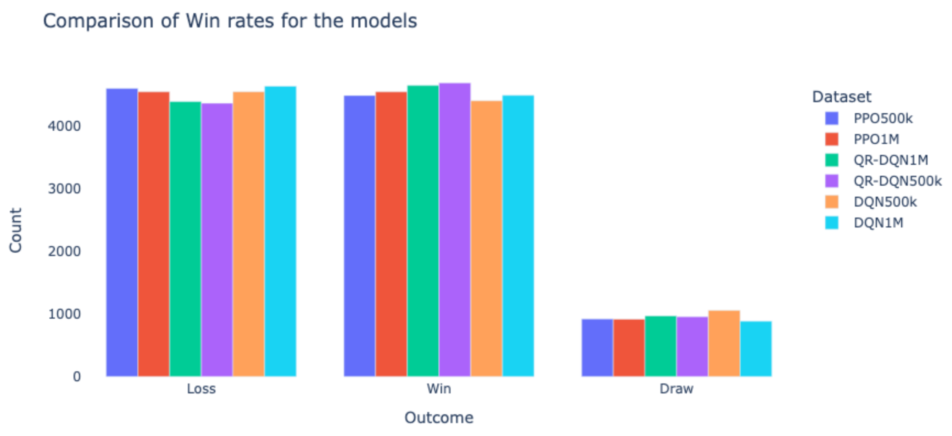

FIGURE 4.2: Models win rates

The plot in Figure 4.2 is crucial to understand the best performing models and thus the most profitable. First of all let us see which models performed poorly, getting more losses than wins. We can see that the PPO models and DQN models got all more losses than wins. The PPO with 500k timesteps and the DQN with 1 million got just slightly more losses than wins, meaning that they either had more luck compared to the other poorly performing models or they were just better in decision making. On the other hand we have the profitable models, both of the QR-DQN got way more wins than losses, achieving a stunning win rate that goes well above the average win rate of a player. Showing that they reached a better decision making process, and almost nullifying the luck factor, since they have got results way better of the other models.
This graph, by the way, just tells us how they performed in the end, but what is crucial is to see which models were effectively learning and which one were just playing randomly. This can be seen using the plot of the win rates over time:

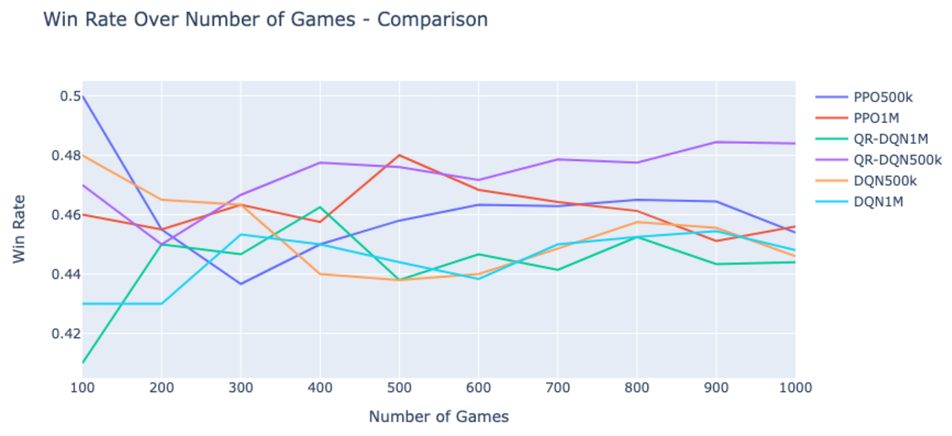

FIGURE 4.3: Models win rates over time

From Figure 4.3 we can instantly identify the models that started lucky and then dropped to the win rate that they were really supposed to achieve. In this category that we will just call "luckies" we can identify the PPO with 500k timesteps and the

DQN with 500k timesteps. They started respectively at 50% and 48% of win rate, to then drop down to 45.5% and 44.6%. This statistics first of all show us that in the short run the true skill of a player does not matter, since the stochastic nature of the blackjack is dominant, but in the long run such randomness gets less and less prevalent, leaving space to the decision making skill of the player.
The PPO with 1 million timesteps is the only one that started and ended at almost the same identically percentage, starting with 46% and ending with 45.6%. This tells us that he probably was lucky during all the games but also showing that he was able to leverage such luck with a good decision making.
We then have the DQN and QR-DQN, both with 1 million timesteps, that started low to get higher and higher, ending way above the starting point. For instance they started with respectively 43% and 41%, ending at 44.8% and 44.4%. This means that probably they were unlucky in the beginning, getting bad cards, but in the long run the decision making is more important and they achieved an higher win rate. Showing once again that in the short run the luck is more important then the decision making, but such situation will flip the other way around in the long run.
Last but not least we have the best performing model, the QR-DQN with 500k timesteps. This model started generally high, with a 47%, and increased overtime, with just one small drop. It ended with the highest win rate between all the models of 48.4%. This means that he was lucky during the start, unlucky after a bit but in general its decision making is really good.

After all these analysis we can now get some conclusions about the best model. Clearly the best model among all of the other models that we have trained is the QR-DQN with 500k timesteps. This model got 48.4% of wins, not considering the draws. Meaning that it lost really a few games and could theoretically be really profitable. Of course such performance comes with a price, that is the long training time, the computational power required and especially the hard theory behind the model that needs to be understood in order to tune it at its best. By the way, all models have performed well if we compare them to the average win rate of a player, showing that they were all able to achieve a good decision making and especially to "understand" the game and the different scenarios.

To conclude we would like to give some general advice regarding the project and gambling in general. The purpose of this thesis and work was to solely prove a point, or at least try to prove it. Gambling can not be profitable in any way, and if it (for any reason) becomes profitable the casinos will instantly change the rules to get the house advantage back. Also, while it is true that one of our models can be seen as profitable, we still have to take into consideration that this is a simplified version of blackjack, and also that it is not legal to bring a laptop at casinos to play with. Regarding online casinos, there is not specific regulations regarding those and the use of AI, but stay assured that if you start to gain some profits they will stop you somehow.
In general the advice is: do not gamble, for any reason. Gambling was never, and never will be, a way to make profit and a living.

Bibliography

[1] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’19. Anchorage, AK, USA: Association for Computing Machinery, 2019, 2623–2631. ISBN: 9781450362016. DOI: `10.1145/3292500.3330701`.

[2] Will Dabney et al. “Distributional Reinforcement Learning With Quantile Regression”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (2018). DOI: `10.1609/aaai.v32i1.11791`.

[3] Will Dabney et al. “Implicit Quantile Networks for Distributional Reinforcement Learning”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1096–1105.

[4] Logan Engstrom et al. “Implementation matters in deep policy gradients: A case study on ppo and trpo”. In: *arXiv preprint arXiv:2005.12729* (2020).

[5] Ashley Hill et al. *Stable Baselines*. `https://github.com/hill-a/stable-baselines`. [Accessed on 29/05/2024]. 2018.

[6] Tom M. Mitchell. *Machine Learning*. Boston, MA, USA: McGraw-Hill, 1997.

[7] Jonas Močkus. *Bayesian Approach to Global Optimization*. Dodrecht, Netherlands: Kluwer Academic, 1989.

[8] Jonas Močkus. “On bayesian methods for seeking the extremum”. In: *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*. Ed. by G. I. Marchuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 400–404. ISBN: 978-3-540-37497-8. DOI: `10.1007/3-540-07165-2_55`.

[9] Ian Osband et al. “Deep Exploration via Bootstrapped DQN”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016.

[10] Arthur Lee Samuel. “Some studies in machine learning using the game of checkers”. In: *IBM Journal of Research and Development* 44.1.2 (2000), pp. 206–226. DOI: `10.1147/rd.441.0206`.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[12] The Gym Library. `https://www.gymlibrary.dev/index.html`. [Accessed on 29/05/2024].

[13] The Gymnasium Library. `https://gymnasium.farama.org`. [Accessed on 29/05/2024].