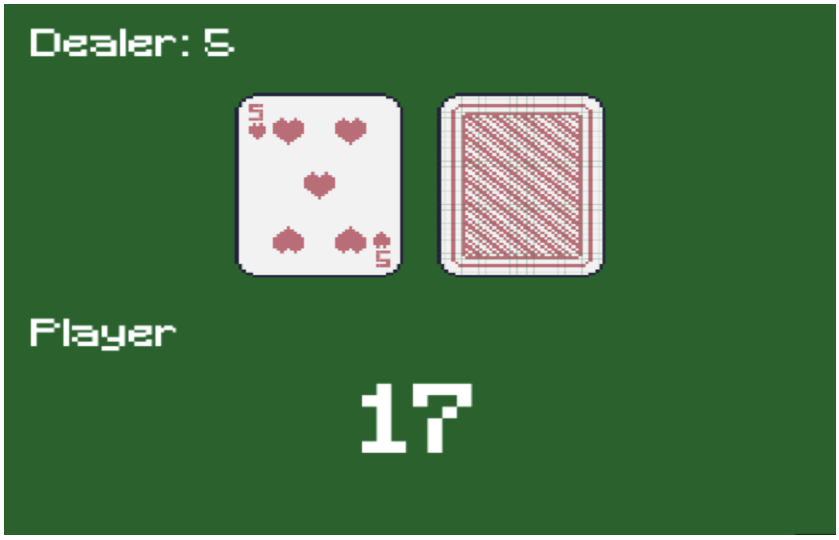
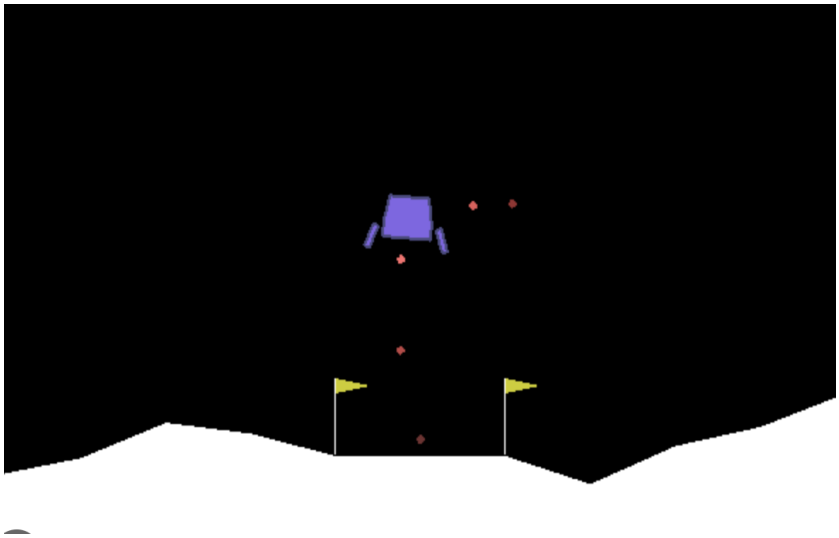


Learn2Learn: Reinforced Gaming Agents



Authors:
Daniel Vidal 1634599
Neil de la Fuente 1630223

Tutor:
Jordi Casas Roma

Autonomous University of Barcelona
November 2023

Contents

1 Introduction

Reinforcement Learning (RL) is a dynamic and powerful paradigm within the field of Machine Learning, focused on designing agents that can make decisions in various environments to maximize cumulative rewards. This report explores the application of RL algorithms to solve different challenges presented by the Gymnasium library.

The project unfolds in three distinct parts, each spotlighting different aspects and applications of RL. The first part delves into tabular solutions within the context of the Blackjack environment, comparing a *Naïve Policy* against the *Monte Carlo* method in search of the optimal strategy. The second segment transitions to approximate solutions, where we apply neural networks and the *REINFORCE* algorithm to tackle the complexities of the Lunar Lander environment. The third and final part of the project conducts a comparative analysis of various RL models applied to the Breakout Atari game, a chosen Gymnasium environment, to offer an in-depth review of their performance and effectiveness.

Each chapter is meticulously crafted to provide detailed insights into the methods and strategies employed, the rationale behind our algorithmic choices, and the nuances involved in parameter tuning. Moreover, the report critically assesses the outcomes of these explorations, complemented by visual data and discussions on the findings and their implications.

Our comprehensive investigation into these RL applications aims to not only demonstrate the versatility of RL algorithms across a range of problems but also to highlight the depth of analysis and understanding required to effectively harness their potential.

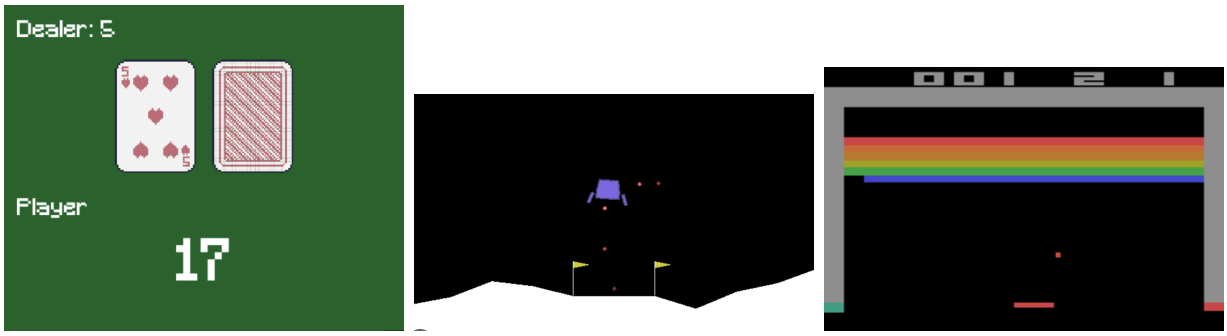


Figure 1: (a) blackjack environment (b) lunar lander environment (c) breakout environment

2 BlackJack Environment - Part 1: Tabular Solutions

This environment is part of the Toy Text environments which contains general information about the environment. 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.

2.1 Description

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 of this game are the following:

- 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 (2-9) have a value equal to their number.

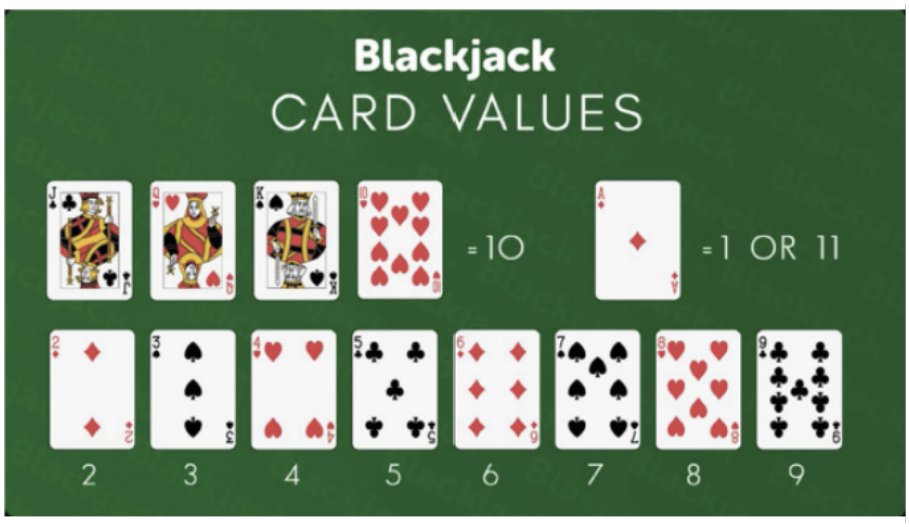


Figure 2: Card Values in Blackjack

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.

This environment corresponds to the version of the blackjack problem described in Example 5.1 in Reinforcement Learning: An Introduction by Sutton and Barto.

2.1.1 Action Space

The action shape is (1,) in the range 0, 1 indicating whether to stick or hit.

- 0: Stick (stop the turn)
- 1: Hit (ask for other card)

2.1.2 Observation Space

The observation consists of a 3-tuple containing: the player's current sum, the value of the dealer's one showing card (1-10 where 1 is ace), and whether the player holds a usable ace (0 or 1).

- The observation is returned as (int(), int(), int()).

Starting State

The starting state is initialised in the following range.

Observation	Min	Max
Player current sum	4	12
Dealer showing card value	2	11
Usable Ace	0	1

Table 1: Starting State

2.1.3 Rewards

- win game: +1
- lose game: -1
- draw game: 0
- win game with natural blackjack: +1.5 (if natural is True) +1 (if natural is False)

2.1.4 Episode End

The episode ends if the following happens:

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

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

2.1.5 Information

No additional information is returned.

2.1.6 Arguments

natural=False: Whether to give an additional reward for starting with a natural blackjack, i.e. starting with an ace and ten (sum is 21).

sab=False: Whether to follow the exact rules outlined in the book by Sutton and Barto. If sab is True, the keyword argument natural will be ignored. If the player achieves a natural blackjack and the dealer does not, the player will win (i.e. get a reward of +1). The reverse rule does not apply. If both the player and the dealer get a natural, it will be a draw (i.e. reward 0).

```
import gymnasium as gym

gym.make('Blackjack-v1', natural=True, sab=False)
```

2.2 Naïve Policy

For this section, the aim is to implement a simple *deterministic Naïve Policy* that chooses the actions according to a simple but not very useful strategy. The strategy consists in:

- The agent will stick if it gets a score of 20 or 21.
- Otherwise, it will hit.

2.2.1 Objectives

With this Naïve agent, the objectives are the followings:

1. Simulate 100,000 games and calculate the agent's return (total accumulated reward).
2. Calculate the percentage of wins, natural wins, losses and draws.
3. Comment on the results.

2.2.2 Implementation of Naïve Agent

The implementation of the Naïve agent is as easier as putting and if-else condition on the policy to check whether the current sum is 20 or more since the policy is deterministic. The policy looks like this:

```python
class NaiveBlackjackAgent:
    def __init__(self):
        pass

    def play(self, obs):
        #stick if player's current sum is 20 or more, else hit
        return 0 if obs[0] >= 20 else 1
```

2.2.3 Results

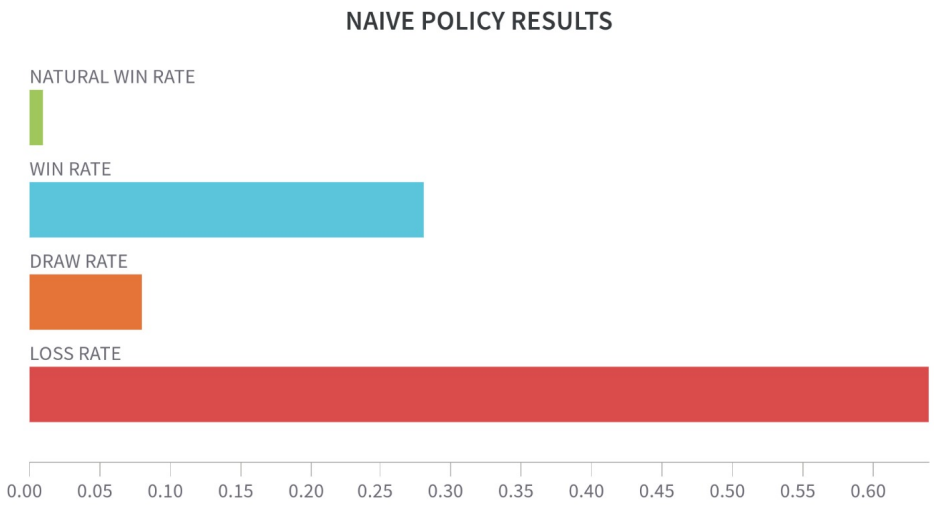


Figure 3: Natural win rate, win rate, draw rate and loss rate of the Naïve Agent

2.2.4 Discussion on Naïve Policy Results

Given that the naive policy achieved a win rate of approximately 27%, its performance can be considered relatively limited. This strategy, which only sticks on a score of 20 or 21, does not take into account the dealer's potential hand, representing a significant strategic oversight. Compared to a baseline random strategy, this win rate does not demonstrate a substantial improvement, suggesting that more sophisticated decision-making processes are necessary for improved performance in the Blackjack environment.

The naive policy's simplistic approach highlights the importance of adaptive learning in reinforcement learning algorithms. Future strategies could incorporate observations of the dealer's visible card, potentially improving the decision-making process. Additionally, employing more complex reinforcement learning algorithms that can learn and adapt to the game dynamics might yield better results.

The statistical significance of these findings is supported by the large number of games

simulated (100,000), indicating that the results are a reliable reflection of the policy's effectiveness and not merely due to chance.

2.3 Monte Carlo Methods

The Monte Carlo method in Reinforcement Learning is a powerful approach for learning the value of states or state-action pairs based on experience, particularly useful in environments like Blackjack where the state space is large and the model dynamics are unknown.

2.4 Monte Carlo with prior knowledge

In this implementation of the Monte Carlo method for the Blackjack environment, it was adopted a purely statistical approach that leverages prior knowledge about the game. Unlike conventional methods that involve playing out entire games by sampling actions from the environment, our method focuses on calculating probabilities based on known game dynamics.

2.4.1 Using Game Knowledge:

In Blackjack, the probability of drawing any particular card can be precisely calculated, as the deck composition and the rules of the game are known. By exploiting this knowledge, it is possible to accurately compute the likelihood of busting for each sum without having to simulate entire episodes from many games.

2.4.2 Computing Probabilities of Busting

The core of this method involves computing the probabilities of busting (exceeding a sum of 21) after drawing a card. This computation is done by drawing a random card for the possible ones for each possible sum that a player can have. This method conducted a total of experiments 100,000 for each sum, to derive these probabilities. For each possible sum that a player can hold, the model calculates the probability of busting if another card is drawn. This calculation considers the values of all possible cards that can be drawn next and the effect of each on the current sum. If the new sum exceeds 21, it is counted as a bust. In the end, the probability of interest is the probability of not busting, which is computed as the number of times the drawing did not bust divided by the total of experiments (100.000).

$$p(s) = 1 - \frac{N^\circ\ of\ bust\ cases}{N^\circ\ of\ cases} = \frac{N^\circ\ of\ not\ bust\ cases}{N^\circ\ of\ cases}$$

The previous is the probability of hit (drawing a card) for the state s, where the state represents a possible sum of cards. With these probabilities, the agent can sample cards given the actual sum of cards to have more chance of making good decisions.

2.4.3 Advantages of This Approach

- **Efficiency:** This method is computationally efficient as it does not require simulating complete games. The probabilities are derived from the known probabilities of card draws.
- **Accuracy:** The use of prior knowledge about the game allows for precise calculations, leading to highly accurate probability estimations. It is known that for the blackjack game, the most optimal that someone can play can lead him to have a win rate of a maximum of around 42% and this approach can obtain a win rate of around 40%.
- **Focus on Critical Decision Points:** By concentrating on the probability of busting, which is a critical decision point in Blackjack, our method provides direct insights into one of the most important aspects of game strategy.

2.4.4 Results

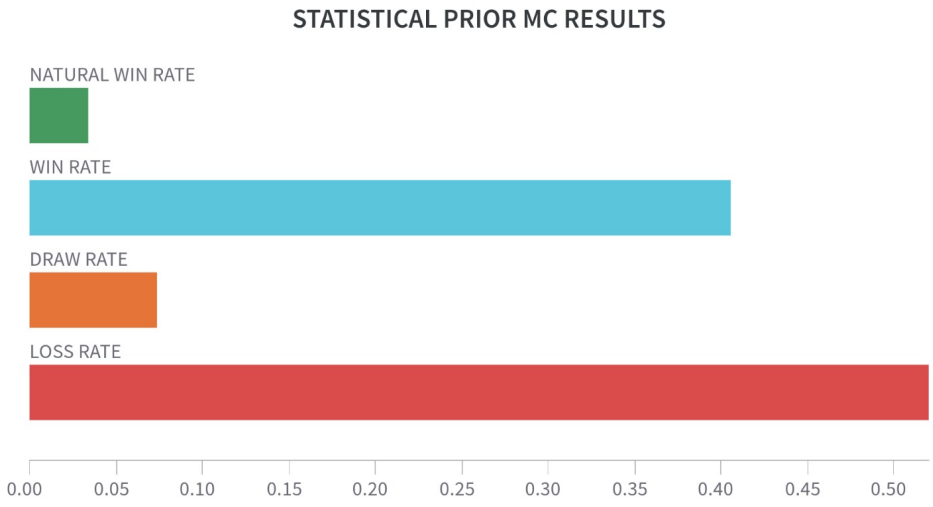


Figure 4: Natural win rate, win rate, draw rate and loss rate of the MC with Prior Knowledge Agent

2.5 First-Visit Monte Carlo

This implementation utilizes the First-Visit Monte Carlo approach. In this method, the value of a state or state-action pair is estimated based on the first time it is encountered in an episode. Through repeated play, or simulation of many games, the average return following each state or state-action pair is calculated, and over time, this average converges to the expected value.

2.5.1 Application to Blackjack

In the context of Blackjack, the state can be represented by a combination of the player's current sum, the dealer's showing card, and the presence of a usable ace. The Monte Carlo method involves simulating numerous games, observing the outcomes, and updating the value estimates for these states or state-action pairs based on the rewards obtained at the end of each game. The characteristic of the Blackjack environment implies that the distinction between first-visit and every-visit Monte Carlo methods becomes irrelevant. In typical scenarios, the first-visit method updates the value estimate based on the first time a state is visited in an episode, while the every-visit method considers every occurrence of the state within the same episode. However, given the improbability of state repetition within a single game of Blackjack, both methods effectively operate identically.

2.5.2 Policy Improvement

Monte Carlo methods are typically paired with policy improvement techniques. After the estimation of state or action values, the policy is enhanced by selecting actions that lead to higher-value states. This iterative process of policy evaluation and improvement continues until convergence to an optimal policy.

2.5.3 Exploration vs. Exploitation

A critical aspect of Monte Carlo methods is the balance between exploration and exploitation. In Blackjack, this was managed through the ε-greedy strategy, where with a small probability ε, a random action is chosen for exploration, and with a probability of 1-ε, the best-known action is chosen for exploitation.

2.5.4 Advantages and Challenges

The Monte Carlo method is straightforward yet effective, especially in environments with stochastic outcomes. However, it requires a significant number of episodes to achieve convergence and might be inefficient in scenarios with lengthy episodes or rare outcomes.

In our application to the Blackjack environment, the Monte Carlo method facilitated the learning of an effective policy for the game. The results from the simulation underscore the learning process's efficacy and the final policy's effectiveness.

2.5.5 Results

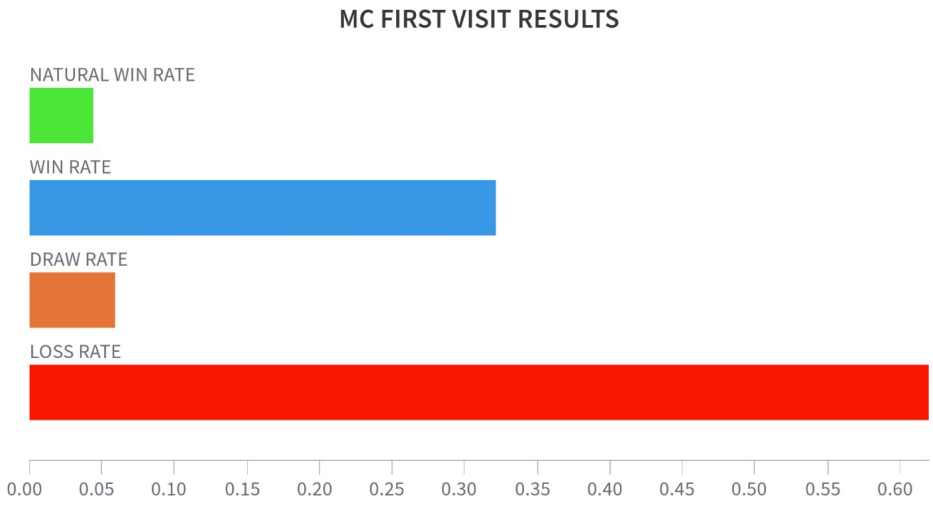


Figure 5: Natural win rate, win rate, draw rate and loss rate of the MC First Visit Agent

2.6 Temporal Differences (TD) Learning Solver Method

Temporal Difference (TD) Learning is a powerful approach in reinforcement learning that combines ideas from both Monte Carlo methods and dynamic programming. In contrast to the Monte Carlo method, which waits until the end of an episode to update value estimates, TD Learning updates estimates based on the current estimate and the observed reward at each step of the episode.

2.6.1 Application to Blackjack

In the Blackjack environment, the TD Solver method was employed to iteratively improve the policy based on each action's immediate outcome, without waiting for the episode's conclusion. This approach allows for quicker learning and adaptation, especially in environments where immediate feedback on actions is available.

2.6.2 Policy Representation

The policy in TD Learning for Blackjack was represented as a value function for each state, which is updated at every time step. The updates depend on the observed reward and the difference between the estimated value of the current state and the next state.

2.6.3 TD Update Rule

The TD update rule used in our implementation follows the formula:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

where $V(S_t)$ is the value of the current state, α is the learning rate, γ is the discount factor, R_{t+1} is the reward received after transitioning from state S_t to S_{t+1}.

2.6.4 Differences from Monte Carlo and Naïve Methods

Unlike the Naïve and Monte Carlo methods, the TD Solver does not require the completion of an entire episode to make updates, leading to faster learning. Furthermore, it effectively balances between the Monte Carlo approach's episodic nature and the dynamic programming method's bootstrapping approach.

2.6.5 Advantages of the TD Solver Method

- **Faster Convergence**: Due to incremental updates, the TD Solver can converge to the optimal policy faster than the Monte Carlo method.
- **Handling of Partial Episodes**: It is effective in environments where episodes do not have a natural end, or in situations where long episodes are truncated.
- **Flexibility**: TD Learning is more flexible and can be adapted to a wide range of environments compared to the Naïve policy.

2.6.6 Results

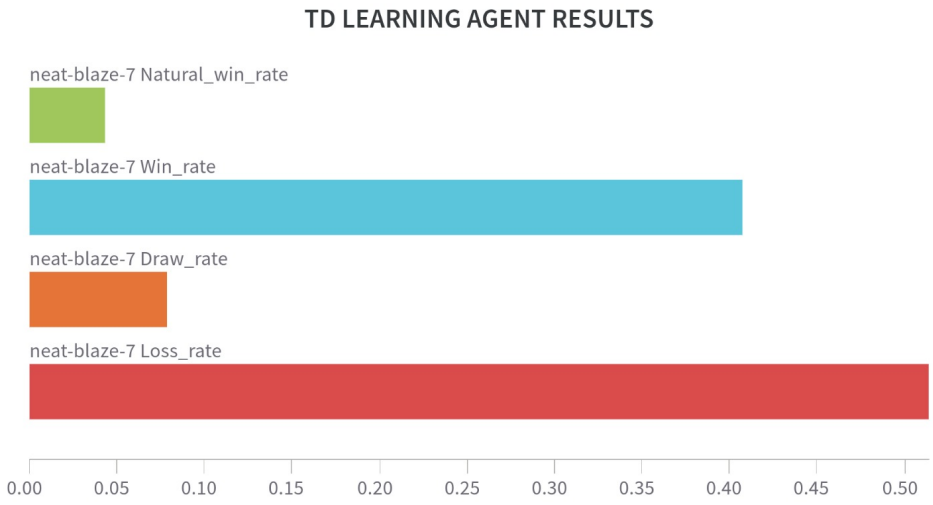


Figure 6: Natural win rate, win rate, draw rate and loss rate of the TD Solver in the Blackjack environment

2.7 Conclusions on Tabular solutions

The results from the tabular solution methods applied to the Blackjack environment provide a compelling view of the performance of various strategies. By evaluating the natural win

rate, win rate, draw rate, and loss rate, we can draw conclusions about the effectiveness of each method.

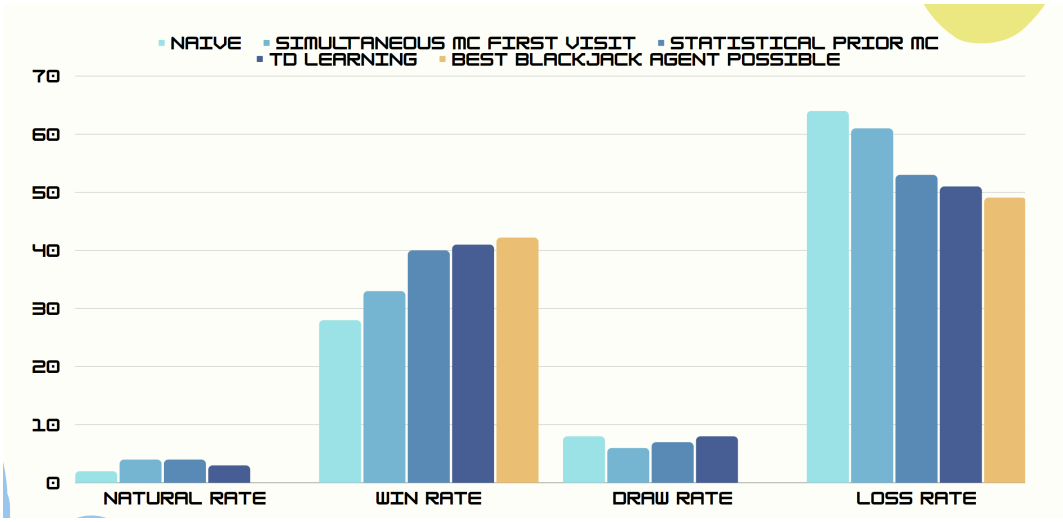


Figure 7: Comparison of natural win rate, win rate, draw rate, and loss rate for Naïve, MC First Visit, Statistical Prior MC, and TD Learning methods against the best possible agent.

2.7.1 Performance Analysis

The Naïve Policy, operating on a rudimentary rule-based system, unsurprisingly registered the lowest performance among the tested strategies. Its limited success aligns with expectations, considering its simplicity and lack of learning capacity.

The Monte Carlo (MC) methods, particularly the Statistical Prior MC, outperformed the Naïve Policy by integrating experience from multiple experiments to refine their policy. The Statistical Prior MC's superiority over the First Visit MC is attributed to its utilization of game-specific statistical knowledge, which allows for more informed decision-making during play.

Temporal Difference (TD) Learning emerged as the top-performing approach, achieving a remarkable 41% win rate. Its success over the other methods can be credited to its ability to learn from each individual step rather than waiting for the end of an episode, as in MC methods. Additionally, TD Learning's method of accounting for both immediate and future rewards creates a balanced strategy that is responsive to the game's immediate states while also being strategic about future potential outcomes. This facilitated rapid adjustments to the policy and better capitalization on the structure of the Blackjack game.

2.7.2 Comparative Insights

When benchmarked against the theoretical "Best Blackjack Agent Possible," which represents an idealized near-optimal performance with a win rate of 42.22%, TD Learning's results were not just an incremental improvement but a significant leap towards this optimal play.

In contrast, while the MC methods showed marked advancements, they did not reach the win rate heights of the TD Learning approach.

In conclusion, TD Learning's capacity to outperform all other models showcases its robustness and potential applicability across a spectrum of reinforcement learning scenarios. Its strength lies in a balanced strategy that continually adapts and learns, making it a prime candidate for tackling complex decision-making tasks that require both precision and adaptability.

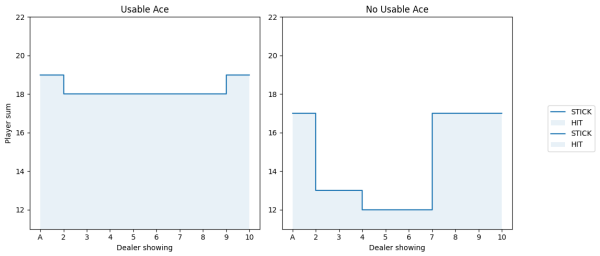


Figure 8: Optimal Policy *(Sutton & Barto, 2018)*

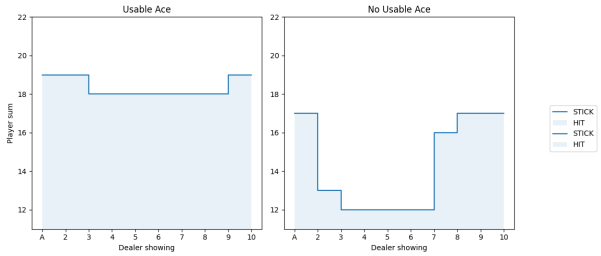


Figure 9: TD Learning Policy

3 Lunar Lander - Part 2: Approximate Solutions

The Lunar Lander environment is part of the Box2D environments. Lunar Lander is a classic rocket trajectory optimization problem that consists of landing a rocket on the moon between two red flags, starting high in the air and trying to maneuver the rocket using engines to propel it so that it lands on its feet without crashing.

3.1 Description

The rocket contains three different engines: the left engine, the right engine and the main engine to propel upwards. According to Pontryagin's maximum principle, it is optimal to fire the engine at full throttle or turn it off. This is the reason why this environment has discrete actions: engine on or off.

There are two environment versions: discrete and continuous. The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector. Landing outside of the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt.

3.1.1 Action Space

There are four discrete actions available: do nothing, fire left orientation engine, fire main engine and fire right orientation engine.

3.1.2 Observation Space

The state is an 8-dimensional vector: the coordinates of the lander in x & y, its linear velocities in x & y, its angle, its angular velocity, and two booleans that represent whether each leg is in contact with the ground or not.

3.1.3 Rewards

Reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives an additional -100 points. If it comes to rest, it receives an additional +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Solved is 200 points.

3.1.4 Starting State

The lander starts at the top center of the viewport with a random initial force applied to its center of mass.

3.1.5 Episode Termination

The episode finishes if:

- the lander crashes (the lander body gets in contact with the moon);
- the lander gets outside of the viewport (x coordinate is greater than 1);
- the lander is not awake. From the Box2D docs, a body that is not awake is a body that doesn't move and doesn't collide with any other body:

3.1.6 Arguments

To use the continuous environment, you need to specify the continuous=True argument like below:

```
import gym
env = gym.make(
    "LunarLander-v2",
    continuous: bool = False,
    gravity: float = -10.0,
    enable_wind: bool = False,
    wind_power: float = 15.0,
    turbulence_power: float = 1.5,
)
```

3.2 REINFORCE Algorithm

In this project, the REINFORCE algorithm, a policy gradient method in Reinforcement Learning, to the Lunar Lander environment. This method is particularly effective for problems where the observation space is continuous or large, as is the case with Lunar Lander.

3.2.1 Algorithm Overview

REINFORCE is a Monte Carlo policy gradient method, meaning it optimizes the policy directly and updates the policy parameters θ at the end of each episode based on the entire sequence of observed rewards.

3.2.2 Implementation Details

- **Network Architecture:** The policy is represented by a neural network. The input to the network is the state of the environment, and the output is a probability distribution over the four actions.

  The network is a sequential model with the following configuration:

  1. A first fully connected layer of 16 neurons and bias=True, with Tanh activation function.
  2. A second fully connected layer of 32 neurons and bias=True, with Tanh activation function.

3. A final fully connected layer with as many neurons as there are dimensions of our action space (one output for each possible action), bias=True, and Softmax activation (dim=-1).

- **Action Selection:** Actions are sampled from the policy network's output distribution, ensuring exploration of the action space.
- **Reward Calculation:** After each episode, the rewards are calculated. These rewards are not immediate but are computed at the end of the episode, characteristic of the Monte Carlo approach.
- **Loss Function:** The loss function is computed as the product of the logarithm of the policy's probability of the taken actions and the cumulative rewards. The policy network's parameters are updated to maximize this function.
- **Training:** The policy network is trained end-to-end, updating its weights using back-propagation and an optimizer like Adam.
- **Stoping condition:** TO avoid the Lunar lander to stay floating in the air forever or trying to get to the middle of the flags very slowly after landing far from them using its right and left engines, we established a limit of steps for episodes in the training.Commonly, the lander would touch down at a considerable distance from the flags and then attempt to get closer by using its lateral engines. This behavior emerged because the reward structure favored the lander's stability upon touchdown, which yielded higher immediate rewards compared to the costlier act of airborne maneuvering using the main engine. To mitigate excessive fuel expenditure and encourage more precise landings, we instituted the step limit, effectively guiding the agent towards more direct and efficient descent trajectories.

3.2.3 Advantages of Reinforce for Lunar Lander

The Lunar Lander environment, characterized by its continuous state space and the need for precise control, poses a significant challenge for reinforcement learning algorithms. The REINFORCE algorithm, which employs policy gradients to directly optimize the agent's actions, is particularly well-suited for this task. It excels in environments where the action's effect is not only immediate but also has long-term consequences on the trajectory of the state.

REINFORCE's strength lies in its ability to adjust policies based on the cumulative rewards, which is essential in the Lunar Lander's dynamic environment. The algorithm's focus on policy gradients enables it to effectively navigate the high-dimensional space by reinforcing actions that lead to successful landings while penalizing less optimal maneuvers. This method of learning is inherently adaptive and can cater to the subtle nuances of the landing sequence, resulting in a refined strategy that consistently achieves the objective of a safe landing.

3.2.4 Results and Analysis

Our implementation of the REINFORCE algorithm demonstrated notable success in training an agent for the Lunar Lander environment. Over numerous episodes, the agent learned to land the craft successfully, balancing exploration and exploitation.

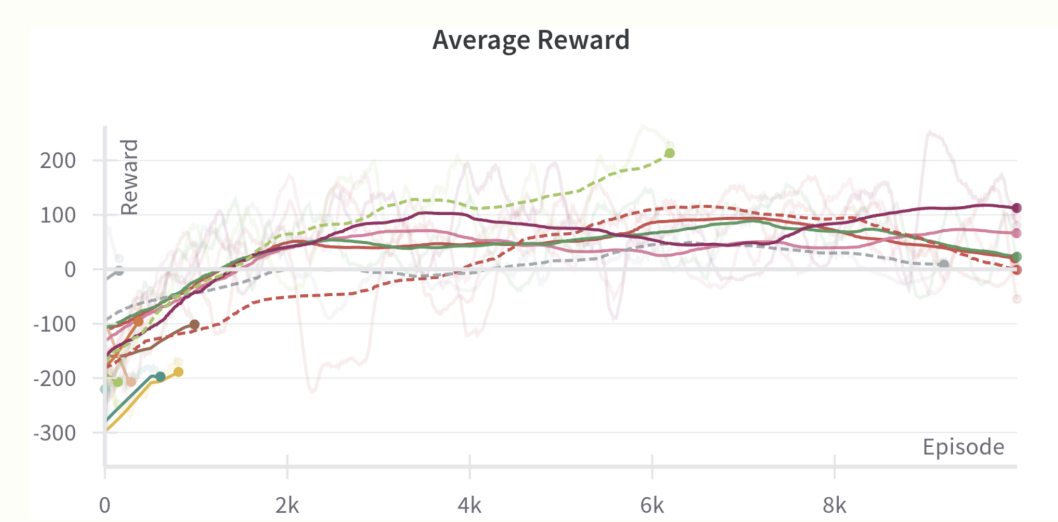


Figure 10: Results of the Reinforce algorithm for several experiments

The application of the REINFORCE algorithm to the Lunar Lander environment has yielded progressive improvement over the course of training. As depicted in the plot of average rewards per episode, initial performance indicated a learning phase where the agent accumulated lower rewards, reflecting a higher frequency of unsuccessful landings.

As the agent continued to experience the environment and learn from its interactions, a notable increase in the average reward was observed, suggesting the development of a more effective landing strategy. The presence of variability in the individual runs, indicated by the lighter lines, is characteristic of the exploration component inherent in policy gradient methods such as REINFORCE.

The stabilization of the moving average at higher episodes signifies that the agent's policy may have converged, demonstrating consistent performance. This trend towards stabilization is a positive indication of the algorithm's capacity to learn and adapt to the task of landing on the lunar surface efficiently.

The results, particularly the agent's stabilized performance, underscore the potential of the REINFORCE algorithm to effectively solve the Lunar Lander environment, provided sufficient time and episodes for learning. The data suggests that the learning process is robust, leading to a policy that can reliably achieve a steady level of performance in this complex task.

In summary, the application of the REINFORCE algorithm to the Lunar Lander environment highlights the effectiveness of policy gradient methods in complex reinforcement

learning scenarios, providing valuable insights into both the algorithm's workings and the dynamics of the environment.

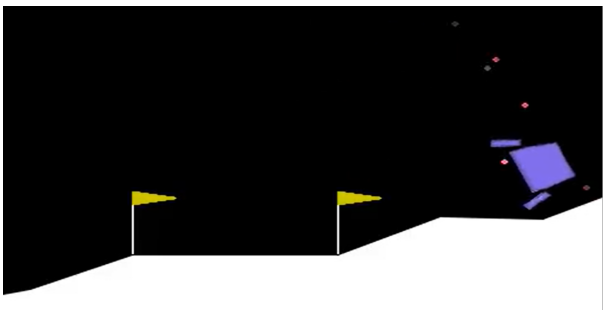


Figure 11: Lunar lander before training

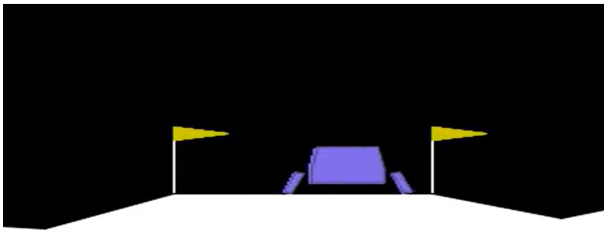


Figure 12: Lunar lander after training

3.3 Conclusions on Lunar lander

The application of the REINFORCE algorithm to the Lunar Lander environment has demonstrated its effectiveness as a policy gradient method in reinforcement learning, particularly suitable for continuous and complex observation spaces. Through the course of training, the algorithm demonstrated a notable capability to incrementally improve the landing strategy, achieving consistent performance as reflected in the convergence of average rewards. This improvement underscores the algorithm's proficiency in balancing exploration with exploitation, a critical aspect of learning in diverse initial conditions.

The REINFORCE algorithm's direct policy optimization approach, without the need for a model of the environment's dynamics, allows for a robust and flexible learning process. The successful adaptation and performance of the agent in the Lunar Lander task suggest that policy gradient methods like REINFORCE can be powerful tools for solving practical problems that require nuanced decision-making and control in environments with continuous observation space.

4 Part 3: Solving a gymnasium environment

The objective of this task was to choose a gymnasium environment to solve and different models from Reinforcement Learning to solve them using either our self implementations or third-party implementations like Stable Baseline 3 (SB3), which is a framework used to develop multiple Deep Reinforcement Learning models to solve environments like the ones provided by the gymnasium library. Then, the idea was to search for the best selection of parameters to get the best results on each model, compare them, select the best model to solve the environment and comment on the results.

For this task we decide to work on the Breakout Atari game from gymnasium. A classical game where the player has to control a paddle moving it to the right or to the left to hit a ball so that the ball does not fall below the paddle and the player must try to break the blocks that are floating above the paddle hitting them with the ball. The player has 5 lives that are lost when the ball falls.

Figure 13: Breakout Atari Game frame

4.1 Description of the environment

The **Observation Space** consists of the current frame in which the player is in the game, the frame is an image of size 210 x 160 x 3 channels, each channel with a pixel value between 0 and 255.

The **Action Space** consists of 4 different actions:

1. No operation (doing noting)
2. Fire (Launching the ball at the beginning of each live)
3. Move to the right
4. Move to the left

The **Reward System** consists of earning points for each block destroyed. The points earned by depend on how high the block is.

We decided to solve this environment because this environment presents a new and interesting challenge to us, different from previous environments, since this environment requires the analysis of frames because the Atari games observations are represented as images. It is also required to analyze many frames at the time to have information of the ball direction movement because with one frame we can only now the position of the ball but not the direction of it.

4.2 Set up of the environment

In order to train our models on this environment we first needed to preprocess the frames given by the environment by converting them to gray-scale images, re-size them to be 84 x 84 and normalize the pixel values to range between 0 and 1.

4.3 Implementation of the models

For the implementation of the model we first decided to do our own implementations for each of the models but since

As we progressed through the implementation phase of various models aimed at solving the specified environment, we encountered a pivotal turning point in our methodological approach. Initially, we aimed to create our own implementations from the ground up. This endeavor, while educational, revealed that it was better to have models implemented by third-party libraries to have more reliable implementations that lead to more accurate comparisons between models that come from the same framework so we can evaluate the most promising models of each of the selected that we are going to mention later.

Our objective was not only to implement models that could solve the environment but also to compare them rigorously, seeking the optimal configuration of parameters for each.

The ambition was to isolate the most efficient model that would not just play the game but excel at it.

Through this process, we discovered the compelling advantages of leveraging Stable Baselines 3 (SB3), a library of high-quality implementations of reinforcement learning algorithms. These implementations have been thoroughly tested and benchmarked, ensuring reliability and performance. Transitioning our models to be based on SB3 offered several immediate benefits: Reliability, benchmarking, consistency, community support, focus on parameter fine-tuning rather than implementaion of the model.

We made a benchmark paper apart to make these comparisons, fine-tuning the models and comment on the results. You can find the document of the paper attached with this document. In that document we explain the different models we selected, the parameters that we tried to optimize, the hypothesis we made on the models before the experiments, the reasons of the behaviors of the models while training, the results obtained and why the the best model outperforms the others. We also include the references and final conclusions in that document.

5 Extra Part: Multigaming Agent

In an extension of our main project, we developed a versatile agent capable of adapting to various environments from the Classic Control section of Gymnasium, namely CartPole, MountainCar, and Acrobot. This agent utilizes a standard Actor-Critic (AC) architecture, known for its flexibility and efficiency across different gaming scenarios.

5.1 Understanding Actor-Critic Architecture

Our implementation of the Actor-Critic model is conceptually analogous to Generative Adversarial Networks (GANs), where the 'actor' is like the 'generator', creating actions, and the 'critic' is akin to the 'discriminator', evaluating these actions. This analogy serves as an intuitive guide to understanding the actor-critic interplay.

$$Actor - Critic \longleftrightarrow GAN$$

$$(Actor : PolicyGeneration) \longleftrightarrow (Generator : DataCreation)$$
$$(Critic : PolicyEvaluation) \longleftrightarrow (Discriminator : DataEvaluation)$$

5.2 Architecture and Mathematical Framework

The Actor-Critic architecture operates on a dual-component structure: the actor proposes actions based on the current state, and the critic evaluates these actions by estimating their value.

Mathematically, the actor updates its policy parameters θ as follows:

$$\Delta\theta = \alpha \cdot \nabla_\theta \log \pi_\theta(a|s) \cdot Q(s,a)$$

Here, $\pi_\theta(a|s)$ denotes the policy function parameterized by θ, representing the probability of taking action a in state s. $Q(s,a)$ is the action-value function estimated by the critic, indicating the expected return of taking action a in state s. The learning rate is denoted by α.

The architecture's adaptability allows for adjustments in the number of inputs and outputs to meet each game's unique requirements, enabling effective training across different environments.

5.3 Performance and Future Potential

While the agent demonstrated success in the CartPole environment, its performance in MountainCar and Acrobot was less successful. We believe that further parameter optimization could enhance the agent's performance across all environments. The success in CartPole suggests the potential of the Actor-Critic model in mastering a range of games.

5.4 Conclusions on the multigaming Actor Critic Agent

In conclusion, our Actor-Critic multigaming agent highlights the architecture's capability in handling complex tasks within various reinforcement learning environments. Its potential for broader applications in reinforcement learning is promising, given its adaptability and effective learning approach.

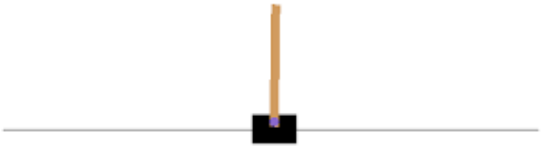


Figure 14: CartPole environment

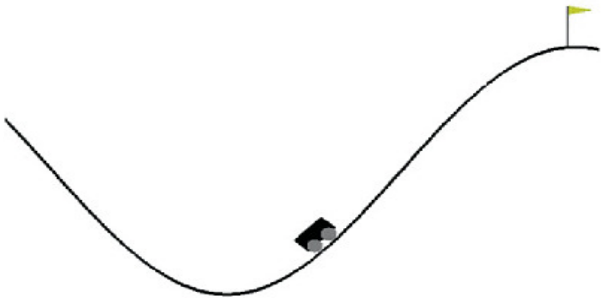


Figure 15: MountainCar environment

Figure 16: Acrobot environment