Universitat Autònoma de Barcelona (UAB)

Bachelor's Degree in Artificial Intelligence

Paradigms of Machine Learning: Final Project

Paula Feliu and Roger Garcia

Lecturer: Jordi Casas Roma

December, 2023

Abstract

This project explores various reinforcement learning techniques in solving complex environments. Initially, we examine tabular solutions like the Naïve Policy and the Monte Carlo method applied to a blackjack environment, highlighting their simplicity and inherent limitations. The project then delves into approximate solutions, notably the REINFORCE algorithm in the Lunar Lander environment, demonstrating the agent's learning progression and policy refinement. Finally, we tackle the Gymnasium's Assault Environment, employing advanced methods like DQN, DDQN, Dueling-DQN, and Actor-Critic models. These methods' efficacy is evaluated in terms of their learning stability and performance in a dynamic, high-dimensional space.

Table of Contents

Chapter 1

Tabular Solutions

In this section, we delve into tabular solutions for reinforcement learning, specifically implementing a naive policy and a Monte Carlo method to train our blackjack agent. These approaches represent the simplest methods for developing optimal policies in reinforcement learning problems, leveraging tables to represent the agent's knowledge and decision-making process.

To effectively utilize these tabular methods, we need to understand the nature of the environment we're operating in. The blackjack environment provides the agent with the current state of the game, consisting of the agent's hand value, the dealer's upcard (the visible card), and the agent's usable ace status (whether the ace can be counted as 1 or 11). Based on this state, the agent decides whether to hit or stand. The environment then simulates the dealer's turn and determines the outcome of the hand, rewarding the agent accordingly.

1.1 Naïve Policy

The naive policy, as its name suggests, is a simple and straightforward strategy that does not involve any sophisticated learning or decision-making. It simply defines a fixed set of actions for each possible state, regardless of the specific circumstances.

The naive policy for the agent is as follows:

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

This policy is deterministic, meaning that the agent always makes the same decision in a given situation. This simplicity makes it easy to implement and analyze, but it is also limited in its ability to adapt to different situations.

1.1.1 Results

The first statistic to state is the return obtained by the agent at the end of the training, with a value of -33030.500, which indicates the expected poor performance of the agent, as this value demonstrates that the agent lost most of the games played. The agent achieved an average win rate of 29.4%, while also demonstrating a respectable natural win rate of 15.97%. The draw rate of 5.94% suggests that the agent is making decisive decisions, leading to clear outcomes rather than ties. With the loss rate value at 64.66%, it indicates that the agent's strategy, while simple, is still not fully optimal. However, it's important to consider that the agent is implementing a simple policy meaning that the results are in accordance with its simplicity.

1.2 Monte Carlo method

In this section, we delve into implementing a Monte Carlo method for training a blackjack agent. Unlike the naive policy, which relies on a fixed set of rules, the Monte Carlo method utilizes a more sophisticated approach to learn optimal strategies by exploiting accumulated experience.

Monte Carlo is a model-free algorithm that estimates the value of states and actions by analyzing past experiences. It does not require explicit knowledge of the environment's dynamics or transition probabilities. Instead, it learns through trial and error, gradually refining its estimates over time.

1.2.1 Epsilon-Greedy Policy

In blackjack, the Monte Carlo method employs an epsilon-greedy policy, which balances exploration and exploitation. With a probability of epsilon, the agent randomly selects an action, allowing it to explore new possibilities and gather additional information. With the remaining probability (1-epsilon), the agent selects the action that maximizes its estimated value, effectively exploiting its knowledge to make informed decisions.

1.2.2 Parameters selection

1. **Number of episodes:**

   Represents the total number of episodes the Monte Carlo control algorithm will run. A higher number of episodes can lead to more accurate policy learning. For this case we used the value of 300.000 episodes as was the one with an assumable computational cost and also good results close to the optimal policy.

2. **Learning Rate (Alpha):**

   A value of 0.02 indicates that, during each update, the agent considers 2% of the new information obtained from the observed rewards. A smaller learning rate means that the agent is more conservative in updating its Q-values and tends to give less weight to the most recent information.

3. **Discount Factor (Gamma):**

   A gamma value of 1 indicates that the agent considers future rewards with equal importance to immediate rewards. This means that the agent fully accounts for the cumulative impact of rewards across future time steps. In this type of scenarios we want the agent to select actions also based on how the episode can turn after that decision because can change the final result of it.

4. **Exploration-Exploitation Trade-off (Epsilon):**

   The parameter 'eps_start' represents the initial value of epsilon, determining the agent's initial willingness to explore. The 'eps_start' is set to 1.0; the agent starts with complete exploration, choosing random actions. Over time will be decreased until the minimum value set for the Epsilon variable, changing the explore behaviour of the agent to exploit behaviour.

1.2.3 Results Analysis

After training the agent using the Monte Carlo method, we evaluate its performance by running it through an extensive set of games, in this case 100.000. For the first of the statistics obtained, the total reward, we can make a clear comparison to the Naïve Policy method. The value in this second case for the return ends up at -4765.00; as the other statistics can confirm, the model wins more than before but it's not able to reach a positive value since the percentage of losses is higher than for wins. The resulting win rate of 42.43% indicates significant improvement over the naive policy's win rate of 29.397%. Additionally from this percentage of wins, the natural win rate is 9.196%, a pretty reasonable value.

The lower draw rate of 8.250% compared to the naive policy's 5.944% suggests that the Monte Carlo agent is making more decisive decisions, leading to clearer outcomes rather than ties. This is further supported by the lower loss rate of 49.317% compared to the naive policy's 64.659%, indicating that the Monte Carlo agent is making better overall choices.

Overall, the Monte Carlo method demonstrates superior performance compared to the naive policy, achieving a higher win rate, a higher natural win rate, and a lower loss rate. This improvement is attributed to the agent's ability to learn from experience and refine its strategy over time.

Chapter 2

Approximate solutions

This part of the project will be focused on the ”Lunar Lander” environment in OpenAI’s Gym, that is a part of the Box2D environments. It is a classic rocket trajectory optimization problem and it is a simulation used in reinforcement learning tasks. It simulates a scenario where a lunar module must land on the moon. The main objective is to safely land the lander between two flags on the surface of the moon. The environment has discrete actions: engine on or off. There are two environment versions: discrete or continuous, but in this case we will use just the discrete version. 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.

2.1 REINFORCE Agent

The REINFORCE algorithm, also known as Monte Carlo Policy Gradient, is a type of policy gradient method used in reinforcement learning. It directly adjusts the policy by estimating the gradient of the expected return (the total accumulated rewards from a state) and uses this to update the policy network’s weights. In the context of the Lunar Lander environment, the REINFORCE algorithm aims to train an agent (the lunar lander) to learn the best actions to safely land between two flags on the moon's surface.

For this project we will use the REINFORCE method with a baseline that is the standardized average of the rewards. First, we will define the neural network model, then we will implement the agent, train it, and finally, we will test the trained agent. The implementation of the REINFORCE agent can be described as follows:

1. **Neural Network Architecture**: The ReinforceNetwork class defines a neural network with three fully connected layers.

   - A first fully connected layer of 16 neurons and bias=True, with Tanh activation function.
   - A second fully connected layer of 32 neurons and bias=True, with Tanh activation function.
   - 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).
   - Then we will use the Adam optimizer to train the network and as the baseline, we will use the standarized average return, computed from the discounted rewards.

   This network, given the current state of the environment, outputs probabilities for each possible action. The tanh activation function helps in normalizing the outputs, ensuring that the network's predictions remain within a manageable range.

2. **Action Selection**: The network employs a stochastic policy for action selection. The get_action method samples an action based on the probabilities output by the network, enabling exploration of the action space.

2.2 Training the agent

Once we have defined our agent, we will pass to the training phase. The process is centered around iteratively improving the agent's policy – its strategy for selecting actions based on the current state of the environment. The training process unfolds over multiple episodes, each representing a complete run of the Lunar Lander environment from start to finish, or otherwise, reaching the maximum steps set at the beginning of the training. In each episode, the agent interacts with the environment, making decisions at each step and receiving feedback in the form of rewards. The goal is to learn a policy that maximizes the total reward over an episode, which in this context translates to successfully landing the lunar module on the moon's surface.

2.2.1 Training process

During training, the agent's neural network predicts the probabilities of each possible action given the current state. Actions are then selected based on these probabilities, allowing the agent to explore different strategies. After each episode, the rewards are processed and used to calculate the policy gradient. This gradient indicates how the policy should be changed to increase expected rewards.

1. **Reward Signal and Discount Factor**: Rewards collected during each step of an episode are used as a signal to guide the learning process. The discount factor (GAMMA = 0.99) discounts future rewards, striking a balance between immediate and future rewards, which is essential in environments where strategic long-term planning is necessary.

2. **Batch Updates and Policy Gradient Calculation**: The model is updated after every few episodes (BATCH_SIZE = 8). The rewards are discounted and the policy gradient is computed. The negative of this gradient is used as

the loss function, guiding the network to adjust its parameters in a way that maximizes cumulative rewards.

3. **Learning Rate**: A learning rate of 0.005 is chosen for the Adam optimizer, ensuring a balanced approach between rapid learning and stability in the training process.

4. **Monitoring and Saving Progress**: The agent's progress is monitored by printing the total reward for every 100th episode (PRINT_INTERVAL = 100) and saving the model when a new best average reward is achieved over the last 100 episodes. This approach helps in tracking the agent's performance and preserving the best model states for future use.

5. **Loss Backpropagation**: After computing the loss, a backward pass is performed, and the optimizer updates the model's weights. This process iteratively improves the policy by reinforcing actions that lead to higher rewards.

2.2.2 Training plots

For a visual view of the training process we have been storing some variables in order to then be able to plot some information to determine the performance of our agent and decide whether the training process was the adequate one or otherwise we needed to change the strategy and readjust some parameters to improve the learning phase.

The first plot we create is the evolution of rewards in training and it can be seen below:

As we can see, the plot is not as understandable as it could be so for a better visualization of the rewards, we plot the same information but as the average rewards every 100 episodes to a clearer view of the evolution.

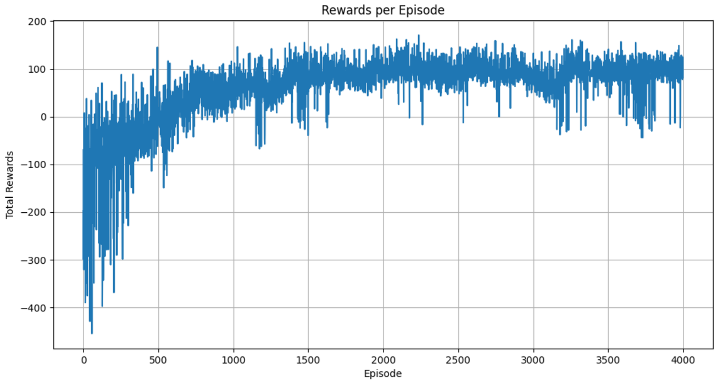


Figure 2.1: Evolution of rewards in training

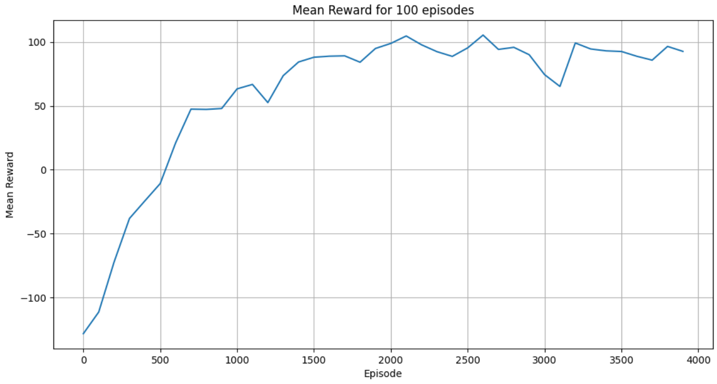


Figure 2.2: Evolution of the average rewards every 100 episodes

In the initial episodes, there's a significant variance in the total rewards, with some episodes reaching as low as -300. This is typical for the early stages of training, where the agent is still exploring the action space and has not yet learned an effective policy.

As the number of episodes increases, the rewards per episode trend upwards, which indicates that the agent is learning from its experiences. The rewards become less negative over time, suggesting that the agent is starting to learn how to avoid actions that lead to large penalties (such as crashing).

Alongside the increase in rewards, the variance in the rewards per episode also decreases. This reduced variance is a sign that the agent's policy is becoming more stable and that it is making more consistent decisions that lead to better outcomes.

Around 2000 episodes, the rewards per episode seem to plateau around a certain level, with fewer instances of extreme negative rewards. This could suggest that the agent has learned a reasonably good policy for the task at hand.

Despite the overall upward trend, there are still fluctuations in the rewards, indicating that the agent continues to explore different strategies within the action space. These fluctuations are crucial for the agent to potentially discover better policies than what it has already learned.

From these plots about the evolution of the rewards, we can infer that the REINFORCE agent has indeed learned over time, as evidenced by the general increase in rewards and decrease in variance. However, the learning process seems to be gradual, and there may be potential for further improvement.

In the next plot, it shows the evolution of the training loss for the REINFORCE agent in the Lunar Lander environment over approximately 500 training intervals.

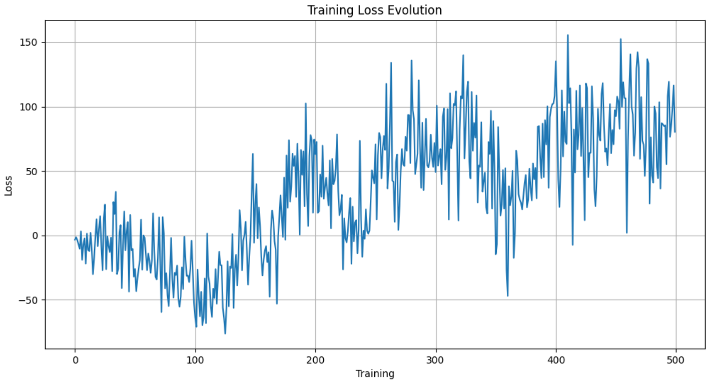


Figure 2.3: Evolution of the loss throughout the training

First of all, we can observe a significant fluctuation in the loss values throughout the training process. This is common in policy gradient methods like REINFORCE, where the gradient estimation can be noisy, leading to variability in the loss.

Unlike typical supervised learning tasks, the loss in reinforcement learning, especially with policy gradients, does not always show a clear downward trend. Instead, the goal is to maximize the cumulative reward, which can sometimes result in an increase in loss if the agent explores new strategies that could lead to higher long-term rewards.

The fluctuating loss also indicates the balance between exploration and exploitation in the agent's learning process. The agent must explore enough of the action space to find good strategies but also exploit known strategies to maximize rewards.

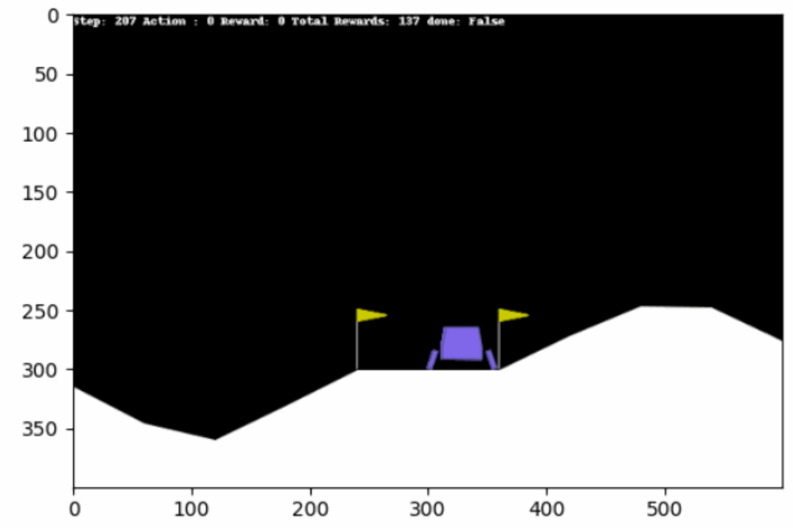
2.3 Testing and results

Once all the training done and we have said the model with the best parameters in the .pth file, we can use this trained agent to evaluate and test our model in the lunar lander environment.

The agent was set to play 100 times, that is 100 episodes or runs and we also set a value for maximum number of steps in each episode.

We have created some gifs of some of the episodes and we will show one of them. In the gif below we can see that the rewards change as the agent selects different actions and they increase as it approaches its final goal:

In case the gif doesnt work, image of the end of the run



```
Step: 207 Action : 0 Reward: 0 Total Rewards: 137 done: False
```

Figure 2.4: Image of the end of the run

2.4 Conclusions

The analysis of the REINFORCE agent's performance on the Lunar Lander environment reveals a significant learning progression. Initially, the agent exhibited a high variance in rewards, a hallmark of exploration, and a foundational phase in reinforcement learning. As training proceeded, there was a marked improvement in the agent's ability to obtain consistently higher rewards. Although a plateau in reward accumulation is noted, it may indicate the formulation of a stable policy, yet with room for potential enhancement through continued learning or parameter tuning.

Furthermore, the training loss plot displays the expected variability inherent in policy gradient methods such as REINFORCE. This variability reflects the agent's ongoing process of exploring the action space, essential for refining strategies and improving policy efficacy. In essence, the agent's consistent achievement of increasing rewards, despite the fluctuations in loss, underscores the successful application of the REINFORCE algorithm. This training has equipped the agent with a robust policy for effectively tackling the complexities of the Lunar Lander environment.

Chapter 3

Solving Gymnasium environment

3.1 Assault Environment

In the realm of reinforcement learning, the Assault environment from OpenAI Gym presents a compelling and challenging testbed. This environment replicates a classic arcade-style game where an agent operates a spaceship with the primary goal of engaging in combat and destroying enemy ships. The graphical interface of the Assault environment is reminiscent of retro video games, providing a vibrant and dynamic visual landscape for agents to navigate. This not only makes the environment engaging but also adds complexity due to the rich visual inputs the agent must interpret.

The action space in Assault is discrete and encompasses a range of movements and behaviors that the agent can execute. These actions include moving the spaceship in different directions, firing at enemies, and combinations of these movements and attacks. The success of the agent hinges on its ability to choose the right actions at opportune moments, balancing offensive maneuvers with defensive strategies to maximize its survival and effectiveness in combat.

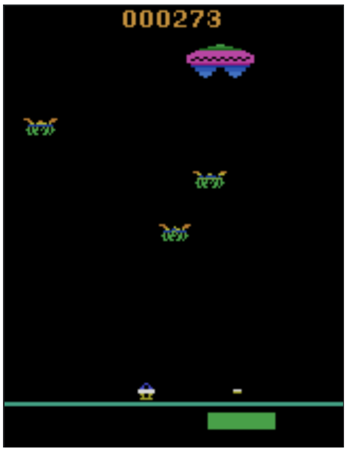


Figure 3.1: Assault environment

In terms of observation space, the Assault environment presents a high-dimensional challenge. The agent receives pixel data representing the current state of the game. This data includes visual cues about the position and movement of enemy ships, the trajectory of incoming fire, and the location and status of the player's spaceship. Interpreting this complex stream of information is crucial for the agent to make informed decisions and adapt its strategy in real-time.

Moreover, the environment provides a nuanced system of rewards and penalties, encouraging the agent to learn effective tactics. Points are awarded for destroying enemy vessels, incentivizing aggressive play. However, the agent must also learn to evade enemy fire, as taking damage can lead to penalties, adding a layer of strategic depth to the environment.

3.2 Approaches

In addressing the challenges of the Assault environment in OpenAI Gym, we have selected a suite of reinforcement learning models and algorithms tailored to its discrete action space. These approaches are chosen for their ability to effectively navigate and learn from the environment's complexities, such as its high-dimensional observation space and the need for strategic action selection.

1. **Deep Q-Network (DQN)**: The Deep Q-Network algorithm represents a significant advancement in reinforcement learning. By combining Q-learning with deep neural networks, DQN can handle the high-dimensional observation space of the Assault environment. Its ability to approximate the Q-value function allows the agent to effectively evaluate and select actions based on the current state. DQN's success in similar environments, like Atari games, makes it a strong candidate for tackling the challenges of Assault.

2. **Double Deep Q-Network (DDQN)**: While DQN significantly improves learning performance, it is prone to overestimation of Q-values. DDQN addresses this issue by decoupling the selection and evaluation of actions, leading to more stable and reliable learning. In the context of Assault, where precise action selection is crucial for survival and success, the stability offered by DDQN can lead to better performance and more consistent learning outcomes.

3. **Dueling Deep Q-Network (Dueling-DQN)**: Dueling-DQN introduces an architecture that separately estimates the state value and the advantage of each action. This structure is particularly beneficial for the Assault environment, where the value of different actions can vary greatly depending on the state. By focusing on learning both state values and advantages, Dueling-DQN can more effectively balance the need for defensive and offensive actions, leading to more nuanced strategies.

4. **Actor-Critic Methods**: Actor-Critic methods combine the benefits of value-based and policy-based approaches. The 'actor' proposes actions based on a policy, while the 'critic' evaluates these actions using a value function. This combination is well-suited for Assault, as it allows for both the direct learning of action policies and the evaluation of action values, enabling the agent to adapt quickly to the dynamic environment of the game.

Each of these models and algorithms offers distinct advantages in the context of the Assault environment. Their selection is based on the need to handle a complex and high-dimensional observation space, the requirement for strategic and timely action decisions, and the overall goal of maximizing performance in a challenging and dynamic setting. The implementation of these approaches will provide valuable insights into their effectiveness and applicability in discrete action space environments.

3.2.1 Hyperparamters

The success of a reinforcement learning algorithm hinges not only on the architecture of the neural network but also on the careful selection of hyperparameters governing the agent's behavior and learning process. In this section, we delve into the key hyperparameters employed in our solution for the 'Assault-v4' environment from the Gym library.

This strategic tuning not only serves to enhance the efficiency of the learning algorithm but also contributes to the adaptability of the agent in navigating the complexities of the target environment.

EPSILON_START, EPSILON_END, EPSILON_DECAY_RATE

These parameters are associated with an epsilon-greedy exploration strategy. Epsilon-greedy is a balance between exploration and exploitation. EPSILON_START represents the initial exploration probability, EPSILON_END is the minimum exploration probability, and EPSILON_DECAY_RATE determines the rate at which epsilon decays over time. Starting with high exploration (EPSILON_START = 1.0) and gradually decreasing it can help the agent to explore the environment initially and then exploit its knowledge later.

EPISODES

This hyperparameter sets the number of episodes during training. The choice of the number of episodes has a huge impact if you want to achieve consistent and representative results. During the project, we faced several problems regarding this parameter because of the limitation in computing resources we had. Not having enough resources limits the learning process of the agent because each episode can take ages.

BATCH_SIZE

It represents the number of experiences sampled from the replay buffer for each training iteration. A larger batch size can lead to more stable updates but may require more memory. 128 is a common choice and generally works well.

GAMMA

This is the discount factor for future rewards. It determines the importance of future rewards in the agent's decision-making process. A high gamma (close to 1) makes the agent consider long-term rewards, while a low gamma makes the agent focus more on immediate rewards.
Our selection, 0.999, means a strong consideration for long-term rewards.

LEARNING_RATE

The learning rate controls the size of the step that the optimizer takes during the update of the neural network weights. 1e-4 is a common starting point. Higher learning rates can lead to faster convergence but might cause instability.

MEMORY_SIZE

This is the size of the replay buffer. A larger buffer allows the agent to store more experiences and can help stabilize training. 10000 is a reasonable size for the situation we are working on. As I mentioned before having computing and memory limitations also affect the selection for this parameter.

OPTIMIZER

Adam (Adaptive Moment Estimation):

In the context of reinforcement learning, Adam is a popular optimizer due to its adaptability and efficiency in handling a variety of optimization landscapes. The algorithm's ability to compute individual learning rates for each parameter is particularly advantageous when dealing with complex, high-dimensional state spaces, as is often the case in reinforcement learning tasks.

Learning Rate (2.5e-4):

The learning rate of 2.5e-4 strikes a balance between the need for rapid convergence and stability. In the 'Assault-v4' environment, this learning rate is chosen to accommodate the potential complexity of the task while avoiding the drawbacks of a learning rate that is either too aggressive or too conservative. The adaptability of Adam, coupled with this learning rate, is well-suited to navigate the environment efficiently.

LOSS FUNCTION

Huber Loss):

The choice of Huber loss is motivated by its robustness to outliers, a crucial characteristic in reinforcement learning scenarios such as the 'Assault-v4' environment. In this dynamic gaming environment, where agent interactions can result in sporadic and sparse rewards, Huber loss ensures that the learning process remains less sensitive to extreme values. This robustness is vital for stabilizing the agent's training and preventing it from being overly influenced by occasional outliers in the reward signal.

By offering a compromise between mean squared error (MSE) and mean absolute error (MAE), Huber loss facilitates a more balanced learning process, a crucial aspect in reinforcement learning where the agent must effectively learn from both immediate and delayed consequences for a stable and efficient learning trajectory.

3.2.2 DQN

DQN is a powerful and adaptive approach that utilizes deep neural networks to tackle the challenges posed by high-dimensional state spaces. Its ability to learn directly from raw sensory inputs, coupled with experience replay and target networks, makes it a robust and widely applicable solution for training agents in complex and dynamic environments.

Architecture

DQN architecture consists of a deep neural network designed to approximate the Q-value function, enabling the agent to make informed decisions. The architecture is defined by the DQN class, which inherits from TensorFlow's Keras Model. Here is a breakdown of the key components:

1. Convolutional Layers: Three convolutional layers (self.layer1, self.layer2, and self.layer3) with batch normalization (self.bn1, self.bn2, and self.bn3). Batch normalization stabilizes training by normalizing layer inputs across batches. This improves the efficiency and stability of feature extraction, particularly beneficial in reinforcement learning tasks with high-dimensional input spaces. There's also a stride of 2 that efficiently reduces spatial dimensions, optimizing computational load and memory requirements

2. Flatten Layer: A flattening layer (self.flatten) transforms the output of the convolutional layers into a one-dimensional vector, preparing it for further processing.

3. Fully Connected Layers: A fully connected layer (self.layer4) with 512 units and ReLU activation captures abstract features from the flattened representation.

The output layer (self.action) is a dense layer with a linear activation function, producing Q-values for each possible action.

Training Process

The training process involves interacting with the game environment, storing experiences in a replay memory, and periodically updating the model. A key aspect of the training is the use of two models: the primary model that learns and predicts actions, and a target model that provides stability to the learning process. The target model's weights are periodically updated with the weights of the primary model to help in converging the learning process.

During optimization, the algorithm samples a batch of experiences and calculates the loss between the predicted Q-values and the target Q-values, which are computed using the reward and the maximum future Q-value estimated by the target model. This loss is then used to update the model's weights through backpropagation.

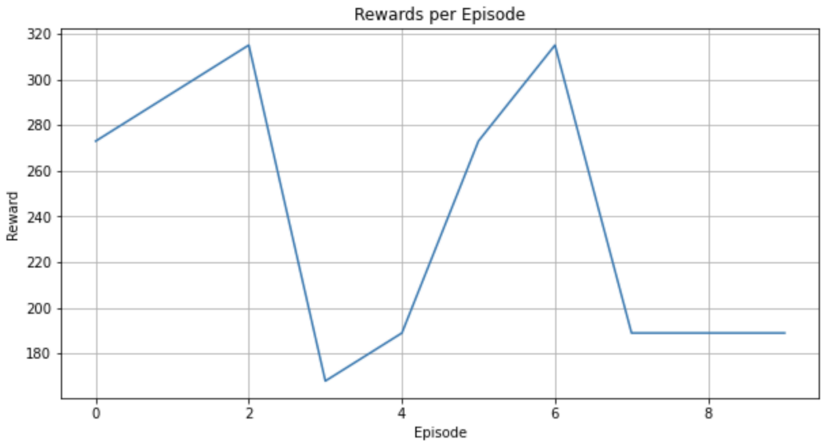


Figure 3.2: DQN training

Figure 3.2 proves that despite DQN is a more sophisticated method than the ones seen until now, the training process is not stable nor optimal. With very unstable learning and low values in rewards evidence the simplicity of the model given the complex environment it needs to face but is a good starting point for more advanced methods.

Nevertheless, during the testing of the best model obtained during training, the agent was able to outperform the previous rewards earned but also in a very unstable process, as you can see in figure 3.3 below.

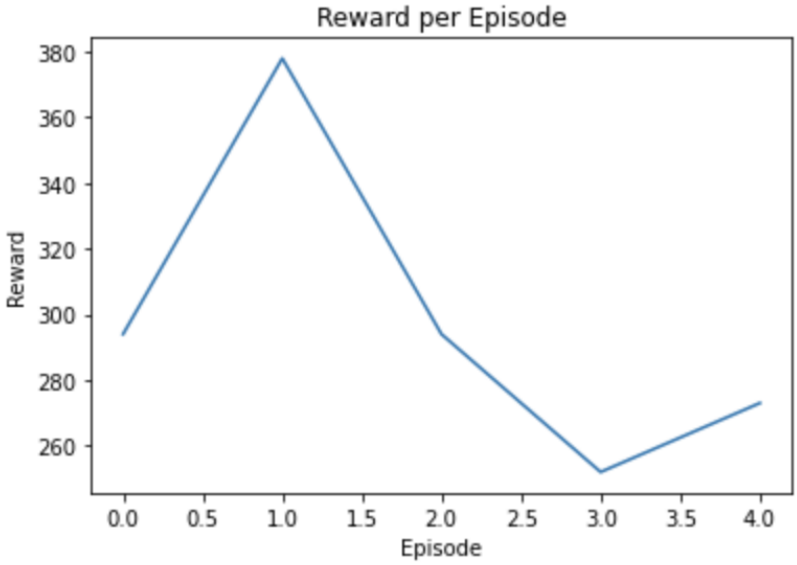


Figure 3.3: DQN test

In conclusion, the DQN model exhibited limitations when applied to the complex Assault environment. This emphasizes the need for more sophisticated algorithms or tailored approaches to effectively address the challenges presented by the environment to achieve optimal performance and robust learning outcomes.

3.2.3 DDQN

The Double Deep Q-Network (DDQN) is an extension of the classic Deep Q-Network (DQN) algorithm, designed to address overestimation bias in action values. While DQN tends to overestimate Q-values, DDQN uses two separate neural networks: one to select actions and another to evaluate those actions. This dual-network structure mitigates overestimation, resulting in more stable and accurate learning.

Architecure

The neural network architecture for both DQN and DDQN remains consistent. The model comprises convolutional layers, batch normalization, and dense layers, culminating in an output layer with linear activation for Q-value estimation.

DDQN Implmentation

The DDQN introduces changes during the optimization process. DDQN calculates Q-values for the next state using the online model, aiming to reduce overestimation. Actions for the next state are selected based on these online Q-values. Q-values for the selected actions in the next state are then computed using the target model. The reduction of overestimation bias is achieved by evaluating the selected actions with a different model (target model). This target model is not updated at every episode, is updated after a fixed number of episodes to give the agent the time to learn. Our update of the target model was every to episodes given the limitations of the computational power. Normally, if you run big amounts of episodes the more episodes you give the more should the agent learn and take better actions.

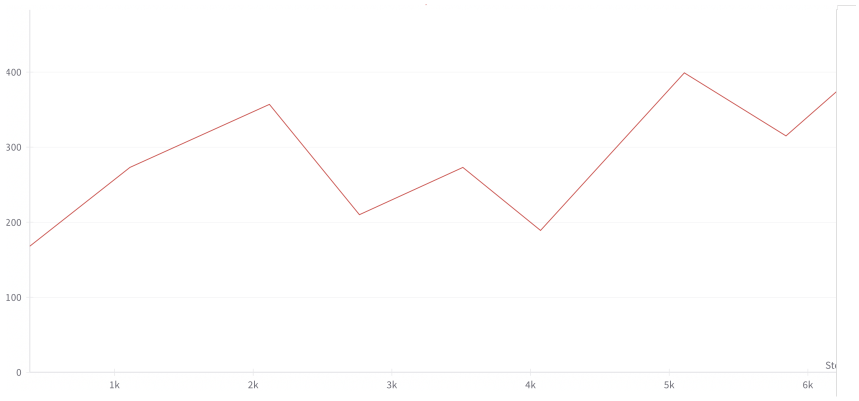


Figure 3.4: DDQN train

Figure 3.4 demonstrates the better performance reached by the DDQN model. As the graph shows the training process is more stable and also has a positive tendency to bigger reward values. During training also a characteristic loss plot was generated as Figure 3.5 shows. These different peaks in the loss value are due to the fact that the target model is updated less frequently compared to the online model, in our case the online model updates every episode and the target every two. During the periods when the target model is not updated, it may fall behind the online model and result in temporarily higher loss values when it is finally updated. Important to say that despite the loss value where increased, in those moments when the target model was updated a bigger reward was earned on the following episode.

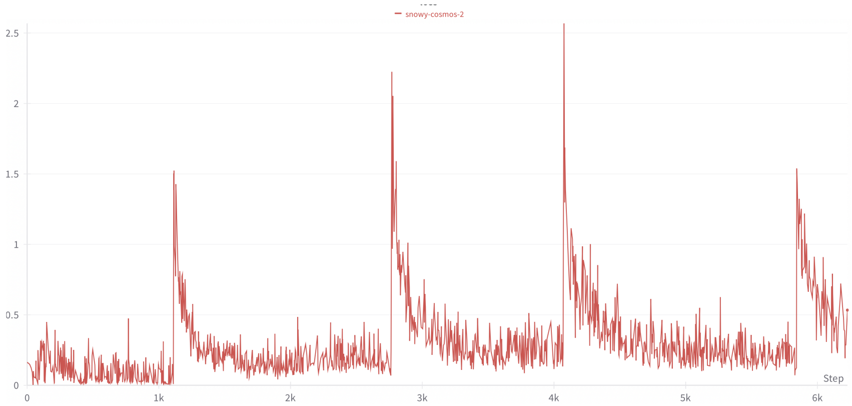


Figure 3.5: DDQN loss plot

On this case and as showed on the previous model, the test process using the best model obtained during training, also has a crazy graph but is important to point out that all reward values were between 300 and 483 which are pretty good results.

3.2.4 Dueling-DQN

The core of this approach lies in the Dueling DQN model. Traditional Deep Q-Networks estimate the value of state-action pairs. However, the Dueling DQN architecture separates the estimation into two distinct components: one for the state value and another for the action advantage. This separation is crucial as it allows for more precise estimates of state value and the advantage of each action, independent of the state.

The model comprises several convolutional layers for feature extraction from the game's visual input, followed by dense layers. The final layers diverge into two paths: one leading to a single neuron estimating the state value and the other to a series of neurons, each corresponding to an action's advantage. These two are then combined to compute the Q-values for each action.

The training process in this algorithm is similar to previous ones. It involves interacting with the game environment, storing experiences in a replay memory, and periodically updating the model. Then we also use the optimization function to sample a batch of experiences and compute the loss using the reward and the Q-values estimated and, therefore with this loss update the model's weights through backpropagation.

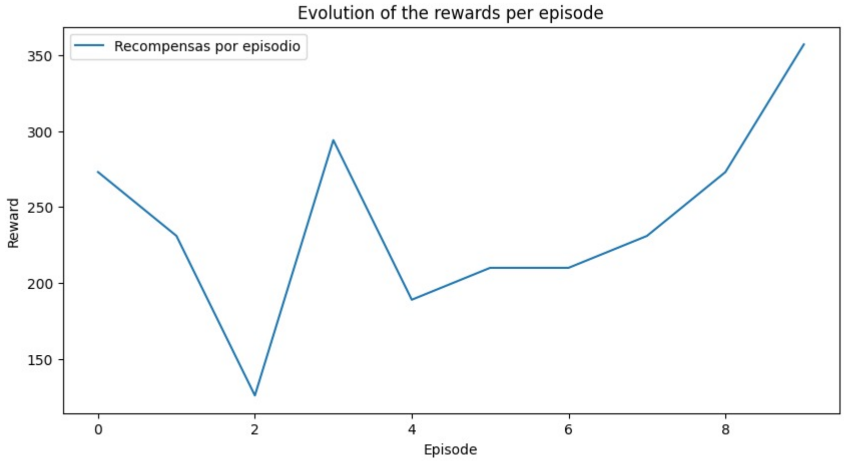


Figure 3.6: Dueling train

The benefits of using a Dueling DQN in this environment are evident from the training results. The architecture's ability to separately estimate the state value and the advantages of each action leads to more efficient learning. This efficiency is reflected in the quicker convergence to higher rewards compared to traditional DQNs and the stability during the training once it reaches the fourth episode.

Throughout the training process, the model's performance is evaluated by tracking the rewards obtained in each episode. As the training progresses, an increase in the average reward per episode indicates the model's improving ability to interact effectively with the game environment. Additionally, the implementation of an epsilon-greedy strategy for action selection ensures a balance between exploration and exploitation, further enhancing the learning process.

In summary, the Dueling DQN presents a significant advancement over traditional DQN architectures by providing a more nuanced understanding of the environment through the separate estimation of state values and action advantages. This leads to more effective learning and better performance in complex environments such as the "Assault" game. The results from this implementation demonstrate the potential of Dueling DQNs in reinforcement learning tasks that require a sophisticated understanding of environmental states and actions.

3.2.5 Actor Critic

The Actor-Critic method is a type of Reinforcement Learning algorithm that combines aspects of policy-based and value-based approaches. It consists of two main components:

- **Actor**: This component is responsible for choosing actions. In our implementation, the actor is a neural network that outputs a probability distribution over actions. The network consists of convolutional layers for processing the game's visual input, followed by dense layers, and it outputs action probabilities using a softmax activation function.

- **Critic**: The critic evaluates the actions taken by the actor by estimating the value function. This neural network, similar in structure to the actor, predicts the value of the current state and is used to critique the actor's performance.

The separation of the policy (actor) and value estimation (critic) allows for more robust learning, as each component specializes in its task.

For the Actor-Critic algorithm, the training processes involves the actor and critic working in tandem. The actor selects actions based on its policy, and the critic evaluates these actions. The experiences (state, action, reward, next state) are stored in a replay memory as in previous algorithms.

During optimization, batches of experiences are used to update both the actor and the critic. The critic is trained to minimize the difference between its value predictions and the actual returns, while the actor's policy is updated towards actions that yield higher returns as per the critic's evaluation.

A notable aspect of our implementation is the use of gradient clipping and learning rate schedules to ensure stable learning.

The Actor-Critic method offers several benefits:

- **Balanced Approach**: By combining policy-based and value-based methods, it leverages the strengths of both. The policy is more exploratory, and the value function provides a solid baseline to evaluate actions.
- **Stability**: The use of a critic for value estimation leads to more stable updates than policy-based methods alone.
- **Efficiency**: The method tends to learn faster and more effectively, as observed in the progressively increasing rewards over episodes in the Assault environment.

Through the course of training, we observed an improvement in the agent's performance in the game environment, as reflected in the increasing rewards. The use of an epsilon-greedy strategy for action selection (balancing exploration and exploitation) further enhances the learning efficiency.

As we can see in the previous plot, with this algorithm we have reached the best results until now and it does reach higher peaks than than the other models at certain points, which may imply that it can exploit the environment's dynamics well when its policy and value function align correctly. However, we can observe that the training process for the Actor-Critic algorithm has not been as stable as the previous one, Dueling DQN.

In conclusion, The Actor-Critic method, with its distinct actor and critic components, offers a balanced and efficient approach to learning in complex environments

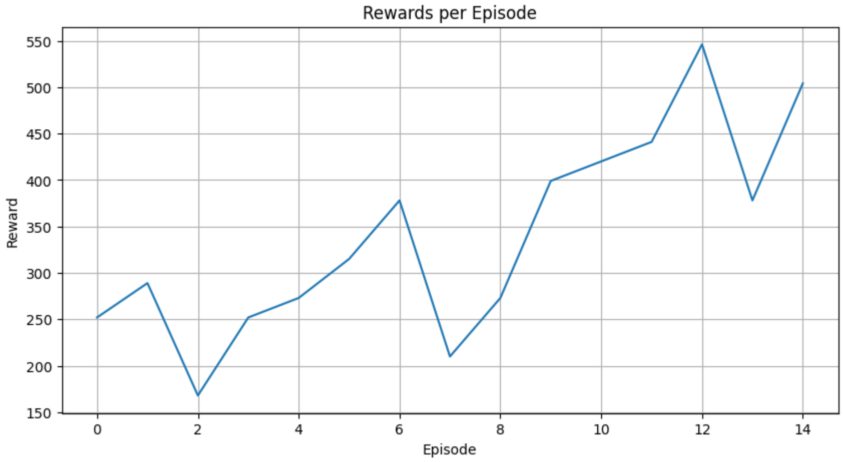


Figure 3.7: AC train

like the "Assault" game. This method's ability to combine policy learning with value function estimation makes it a robust choice for training agents in environments with high-dimensional input spaces and a large number of potential actions. The results from this implementation show promise for the application of Actor-Critic methods in similar reinforcement learning tasks.

3.2.6 Optimization through conversion to gray scale in Actor Critic model

In an effort to optimize the training process and reduce computational cost in our actor-critic model, we have implemented a strategy of converting game frames from RGB to grayscale. The main goal of this modification was to enable more efficient training and the ability to process a higher number of episodes in a shorter computational time. At first, this adaptation was considered promising, as it would allow for a quicker and more agile analysis of the gaming environment, while maintaining the essence of the visual information necessary for the agent's learning.

However, after implementing this modification and conducting a detailed follow-up of the training process, we observed a decrease in the model's performance compared to its original version that processed color images. Despite this reduction in efficacy, it is important to note that the model adapted to grayscale still proved to be promising, achieving competent results compared to previous versions. This observation was evidenced through various performance metrics and visualizations of the learning process, as shown just below, where a slightly flatter and also more unstable learning curve for the grayscale model can be seen, with many high and low spikes but without a constant positive trend of improving rewards as the agent trains.

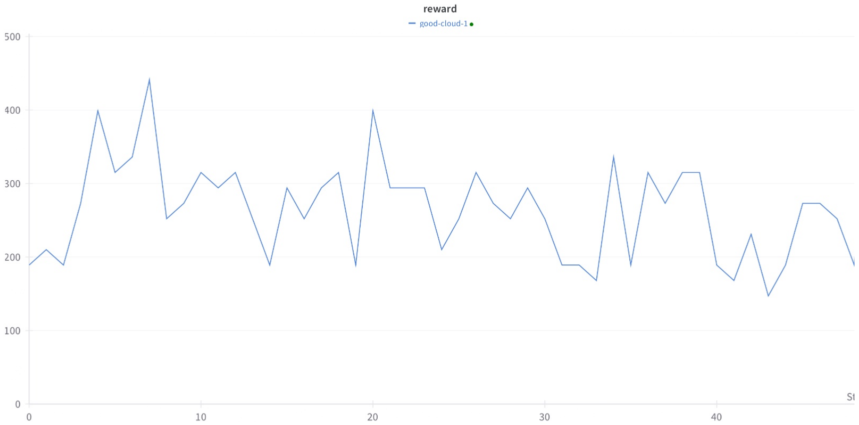


Figure 3.8: AC grayscale train

With an analysis, we believe we have identified several potential reasons behind this decrease in performance. Firstly, the conversion to grayscale entails an inevitable loss of information, since the color dimension of the input data is eliminated. In games like 'Assault', where certain colors may play a crucial role in distinguishing elements and situations, this reduction in information can be significant. Additionally, visual features that are prominent in the color format may be lost or become less distinguishable in grayscale, which potentially affects the model's ability to make accurate predictions or learn effectively. Finally, the alteration in the representation of the frames also implies a change in the distribution of the input data and can affect the agent's performance during training.

These conclusions underline the importance of considering the balance between computational efficiency and the integrity of visual information in the design and optimization of reinforcement learning models.

3.3 Comparison and conclusions

The following plot offers a comprehensive visual comparison of the training processes for the four reinforcement learning models: DQN, DDQN, Dueling DQN, and Actor-Critic, as applied to the "Assault" game environment. To ensure a fair and accurate comparison, each model underwent the same training regimen, with adjustments made to align their learning conditions as closely as possible. This level of standardization across the training processes allows us to observe the unique learning dynamics and performance characteristics of each model. By presenting their training trajectories jointly, we can discern how each algorithm navigates the complexities of the environment, how quickly and steadily they improve, and how their reward patterns evolve over time. This side-by-side visualization aims to highlight the distinct advantages and limitations inherent in each approach, providing clear insights into their comparative effectiveness in a controlled setting.

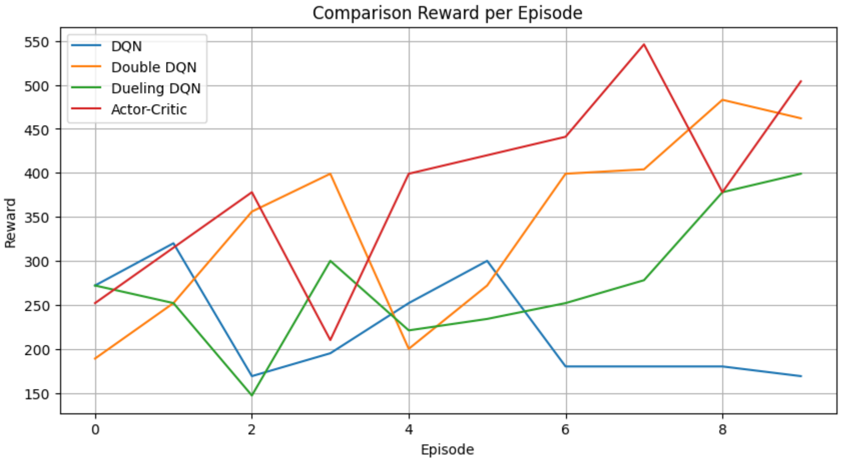


Figure 3.9: Assault environment

We observe that initially, the DQN model provided a robust foundation, mastering high-dimensional state spaces through deep learning techniques. It introduced stability via experience replay and target networks, but was prone to overestimating Q-values. The DDQN model sought to overcome this by separating action selection from evaluation with its dual-network architecture, enhancing stability and accuracy. This last model has reached pretty high rewards in specific episodes but it has had an inestable learning.

The Dueling DQN model represented a significant innovation by decomposing the Q-value function into separate streams for state value and action advantage, leading to more precise learning signals and a pattern of stable, consistent rewards. Contrastingly, the Actor-Critic model showed dynamic learning potential with its policy-based actor and value-based critic, though it experienced greater variance in rewards, suggesting a need for refined balance between exploration and exploitation.

In summarizing the performance and learning stability of these models, Dueling DQN emerges as a particularly balanced option, achieving steady reward improvement while maintaining learning consistency. The Actor-Critic model, while capable of reaching higher reward peaks, demonstrated variability that could benefit from further model tuning to enhance training stability. Ultimately, the choice between these models is contingent upon the specific objectives of the learning task, with Dueling DQN favored for reliability and Actor-Critic offering potential for high-yield returns with appropriate adjustments.

Chapter 4

Conclusions

In the blackjack environment, we applied basic tabular methods such as the Naïve Policy and Monte Carlo approach. These methods proved to be a solid foundation for understanding reinforcement learning's core principles. The simplicity of the Naïve Policy offered a straightforward introduction, while the Monte Carlo method provided a deeper insight into policy iteration. However, their performance was limited by the complexity of the environment and the inability to scale with increasing state spaces.

The application of the REINFORCE algorithm in the Lunar Lander environment marked a significant advancement from tabular methods. This part of the project demonstrated the algorithm's ability to learn and adapt to a more complex, continuous space. The progression and refinement of the agent's policy over time underscored the effectiveness of policy gradient methods in environments with higher dimensionality and more nuanced control requirements.

The Assault environment was an excellent testbed for comparing the Deep Q-Network (DQN), Double DQN (DDQN), Dueling-DQN, and Actor-Critic models. This comparison highlighted the strengths and weaknesses of each model in a dynamic and visually complex environment. DQN provided a robust baseline, while DDQN offered improvements in learning stability. Dueling-DQN's architecture enhanced state-value estimation, and the Actor-Critic model demonstrated a balanced approach between value and policy-based methods. The comparison provided valuable insights into selecting and optimizing reinforcement learning algorithms for various tasks.