Universitat Autònoma de Barcelona

School of Engineering

Joan Lafuente & Adrián García
1523726 & 1634605

Machine Learning Paradigms Project

Submitted for the degree of Artificial Intelligence

December 2024

1 Frogger ALE environment

1.1 Description of the environment

For the "simple" ale game, we have decided to use Frogger. This game consists of moving a frog and making it cross a highway, avoiding the cars, and a river using the objects. Once the frog has crossed both obstacles, it has to be situated on one out of the five squares located at the other side of the river. Also, when the frog has reached one of the final possible destinations a new one appears which has the same objective, but this time one of the squares is already occupied. This happens iteratively until five frogs have crossed both obstacles and all the final locations are used, and the player wins the game. An Image of the game, can be seen at Figure 1.

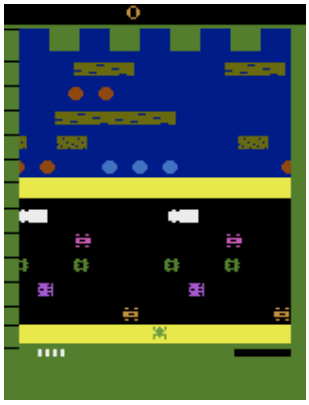


Fig. 1: Image of the initial state of Frogger.

The maximum reward on this game is 400, in the case in which the 5 frogs have crossed the road, river and each one of them are located at one of the five possible positions, this locations cannot be repeated.

Note that the actions that the agent can take in the environment are discrete, specifically 5. Below, that is the table to action indices and meaning:

- **0**: NOOP
- **1**: UP
- **2**: RIGHT
- **3**: LEFT
- **4**: DOWN

1.2 Prepossessing

The prepossessing used at the first part of this project, for Frogger game, consists of four steps. We first make the frame skipping setting, we have used a value of 16, decided after experimentation with our simpler DQN algorithm. There is a justification on the parameter search section of Section 1.3. Then we resize the images to a size of 84x84. Finally, we stack the last 4 observations, in other to have a sense of movement, and scale the image in the range [0, 1]. This setting has been used for both DQN algoriithms.

1.3 DQN with some extensions

Introduction:

The first algorithm we have implemented to play Frogger has been a basic version of DQN, which contains some extensions. This has been done to establish the results that can be achieved with a "simple" version of DQN and then be able to compare it with a more complex algorithm.

This first version already contained three extensions, Double DQN, Dueling and Prioritized Experience Replay sampling. The advantages provided by each of the extensions are the following:

- **Double DQN:** Fixes the overestimation problem with a modification on the loss. To compute the target value, we use the target Q-function and the action predicted by the Q-function being trained.
- **Dueling DQN:** Improves training stability and produces faster convergence, thanks to a separation of the value function and the advantages of each of the predicted actions on the network architecture.
- **Prioritized Experience Replay:** Improves convergence and the policy quality. It is made by making more probable to sample examples with higher errors than those well predicted, improving the sample efficiency.

Parameter search:

On the hyperparameter search we have tested different values for the learning rate and for the discount factor. We fixed the rest of hyperparameters.

For the learning rate comparison, we have used three possible values (0.001, 0.0001, 0.00001). From the experiments we see that using a learning rate of 0.001 or 0.00001 does not allow the model to learn. This is caused for different reasons on the higher value we may be overshooting on the loss landscape, making it impossible for the model to learn. On the case of the low learning rate the model might be getting stucked on local minima or learning too slow to be able to see a clear evolution at the rewards obtained. This comparison was made using a discount factor of 0.99, and can be seen in Figure 2.

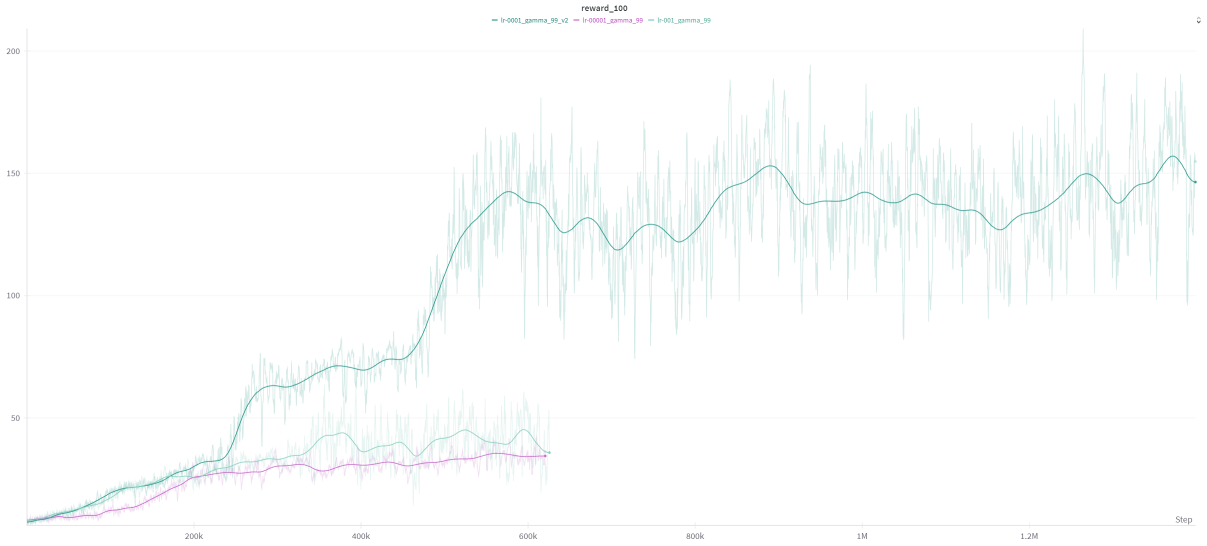


Fig. 2: Comparison of the rolling average reward, 10 episodes, evolution during training for different values of learning rate. The three experiments use the same discount factor of 0.99.

Once we determined the best learning rate value, 0.0001, we experimented with different values for the discount factor. We tested the use of a lower value of 0.9. At Figure 3 we can see that reducing the discount factor improved the convergence speed, by making it easier for the model to increase the expected reward. This means that to play well on this game are more relevant the rewards given shortly after an action than those more separated in time. This can be produce as taking a wrong action at the game might make you completely fail at the following time step, making more relevant the short term effects.

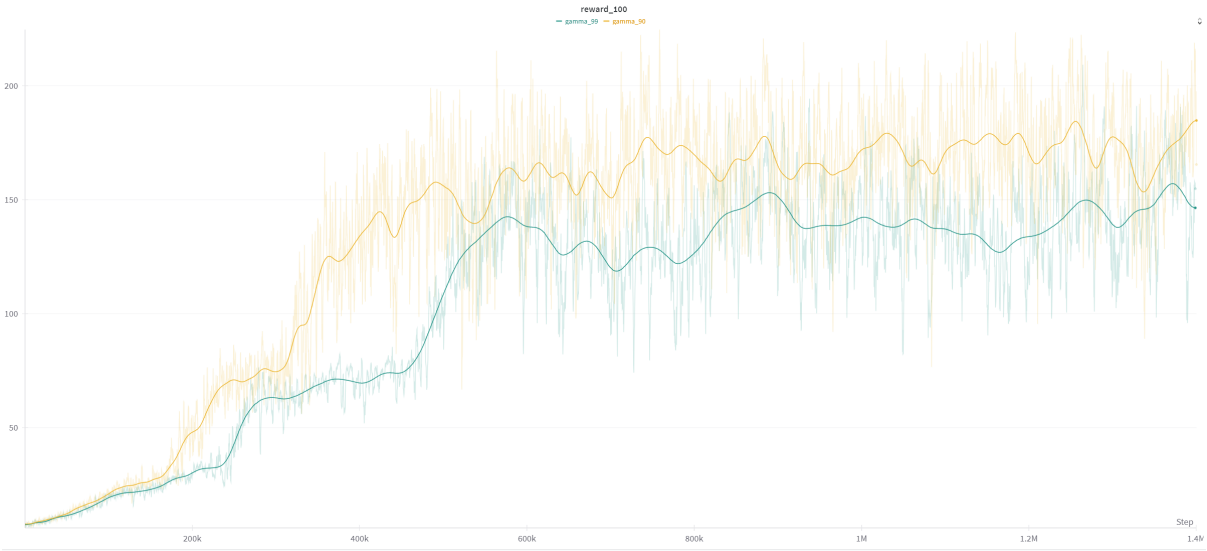


Fig. 3: Comparison of the rolling average reward, 10 episodes, evolution during training for different values of discount factor (Gamma). The two experiments use the same learning rate of 0.0001.

It is important to notice that none of our experiments were capable of solving the game, we think that it can be caused due to not having an enough number of training steps. We say this as during the last time steps, some episodes already reached a reward of 400. We did not try with larger values because of the training time required, as to achieve this results it already took more than 8 hours.
Another detail is that we have used a not usual frame skipping value of 16. If lower values were used, such as 4, the model was not capable of progressing on the game. We consider this might be caused because many images may not provide useful information, the cars and objects follow the same trajectories and speeds always. Also, it can help because it helps seen more different situations quicker, improving training speed.

Results and visualizations:

We can see that our algorithm allowed our model to learn correctly, reaching average rewards of over 200 during training. This translates into our policy being able to cross the road and river on average with 3 out of the 5 five frogs. There is a video of the behavior on the folder specific to this algorithm at the public GitHub repository of this project.

When testing our model, over 100 episodes, we see an average reward of 265.44, with a standard deviation of 128.07. This means that a significant amount of times is solving the game, as the maximum achievable reward is within the standard deviation. For this reason, we consider that using more training steps would allow the model to exploit more the information learned, and in consequence solving the game more often.

1.4 Rainbow DQN

Introduction

As for the second model implementation, we have decided to create a version of the **Rainbow DQN** algorithm. Note that this **Rainbow DQN** implementation integrates the main components to solve the *Frogger environment*. Know that the training loop combines **N-Step Prioritized Replay**, **Noisy Layers**, **Duelling DQN**, and **Double DQN** to improve exploration, stability, and learning efficiency. Below, we highlight the critical details of the training process:

It must be commented that this particular implementation of the Rainbow algorithm does not include the Categorical DQN and we acknowledge that further implementation of it would benefit the current implementation

In the following sections we discuss the main *implemented novelty*, the logic behind the algorithm, the **problem** we encountered and, how the proposed **solution** has the potential to solve it long-term.

Exploration vs. Exploitation Tradeoff

- **Noisy Layers:** Exploration is handled using *Noisy Networks*, where noise is injected into the network's linear noisy weights during training.
- Each layer learns two parameters: μ (mean) and σ (standard deviation), which are optimized during training.
- *Initially*, the standard deviation σ dominates, causing the agent to explore widely. As training progresses, σ decreases, and the network starts exploiting learned behaviors.
- **Gaussian Noise Injection:** To further address the problem of local minima, Gaussian noise is added dynamically to the noisy layers when a plateau is detected in the mean reward.

Plateau Detection and Gaussian Noise Injection

- The algorithm monitors the recent mean rewards over a sliding window of length L_{plateau}.
- If the reward improvement falls below a predefined threshold τ_{plateau}, the system detects a plateau.
- **To escape local minima:**
  - Gaussian noise is added to the parameters of the noisy layers:

$$\sigma_{\text{weights}} \leftarrow \sigma_{\text{weights}} + \sigma_{\text{factor}} \cdot \mathcal{N}(0, 1).$$

- This perturbation encourages the agent to explore new actions, potentially discovering better strategies.
- Over time, the noise is decayed using a factor σ_{decay}, restoring the balance between exploration and exploitation.

Prioritized N-Step Replay Buffer

- The replay buffer stores transitions in the form $(s_t, a_t, r_t, d_t, s_{t+n})$, where rewards are accumulated over n-steps to accelerate learning.
- Transitions are sampled based on their **TD-error** δ, which measures the difference between the predicted and target Q-values:

$$\delta = |Q_{\text{current}} - Q_{\text{target}}|.$$

- Higher TD-error experiences are assigned higher priority, ensuring that the agent focuses on learning from the most *informative* transitions.
- Importance-sampling weights are used to correct for the bias introduced by prioritization.

Double DQN and Duelling DQN

- **Double DQN:** Reduces the overestimation bias by using the target network to evaluate the Q-values of the best action selected by the online network:

$$Q_{\text{target}} = R^{(n)} + \gamma^n Q'(s_{t+n}, \arg\max_a Q(s_{t+n}, a)) \cdot (1 - d^{(n)}).$$

- **Duelling DQN:** Decomposes the Q-values into **state-value** and **advantage** components, allowing the network to learn which states are valuable without needing to know the best action in those states.

Periodic Target Synchronization

- To stabilize training, the target network Q' is synchronized with the online network Q at regular intervals T_{sync}:

$$Q' \leftarrow Q.$$

*On the following a **complete implementation** of the algorithm can be seen:*

Algorithm 1 Rainbow DQN with N-Step Prioritized Replay, Duelling DQN, Double DQN, and Noisy Layers

Input: Environment $\mathcal{E}$, replay buffer $\mathcal{B}$, discount factor γ, networks Q, Q'
Plateau threshold τ_{plateau}, plateau length L_{plateau}, patience P_{patience}
Noise parameters: σ_{factor}, σ_{decay}, duration T_{noise}
Sync interval T_{sync}, maximum steps $T_{\max}$, batch size N_{batch}, N-steps n
Initialize: $Q, Q' \leftarrow Q$, $t \leftarrow 0$, $\eta \leftarrow$ False, $p \leftarrow 0$, $\mathcal{R} \leftarrow []$
while $t < T_{\max}$ **do**
    $t \leftarrow t + 1$
▷ Plateau Detection and Noise Injection:
    if is_plateau$(\mathcal{R}, \tau_{\text{plateau}}, L_{\text{plateau}})$ **and** $\eta =$ False **then**
        $\sigma_{\text{weights}} \leftarrow \sigma_{\text{weights}} + \sigma_{\text{factor}} \cdot \mathcal{N}(0, 1)$
        $\eta \leftarrow$ True, $t_{\text{noise}} \leftarrow t, p \leftarrow 0$
    else if η **and** $t - t_{\text{noise}} > T_{\text{noise}}$ **then**
        $\sigma_{\text{weights}} \leftarrow \sigma_{\text{weights}} \cdot \sigma_{\text{decay}}, \eta \leftarrow$ False
    end if
▷ Environment Interaction:
    Select action $a_t \sim Q(s_t, a; \sigma_{\text{weights}})$, observe r_t, s_{t+1}, d_t
    Store transition $(s_t, a_t, r_t, d_t, s_{t+1})$ in $\mathcal{B}$ with priority $\delta = |Q_{\text{current}} - Q_{\text{target}}|$
    if $|\mathcal{B}| < N_{\text{buffer}}$ **then** *Skip iteration*
    end if
▷ Prioritized N-Step Replay and Loss Computation:
    Sample $(s, a, R^{(n)}, d^{(n)}, s_{t+n}) \sim \mathcal{B}$ with priority weights w
    Compute N-step target Q-values:

$$Q_{\text{target}} = R^{(n)} + \gamma^n Q'(s_{t+n}, \arg\max_a Q(s_{t+n}, a)) \cdot (1 - d^{(n)})$$

    Compute current Q-values: $Q_{\text{current}} = Q(s, a)$
    Compute prioritized loss:

$$L = \mathbb{E}\left[w \cdot (Q_{\text{current}} - Q_{\text{target}})^2\right] + \lambda \cdot \sum |\sigma_{\text{weights}}|$$

    Update Q via backpropagation and update priorities in $\mathcal{B}$
▷ Periodic Target Synchronization:
    if $t \mod T_{\text{sync}} = 0$ **then** $Q' \leftarrow Q$
    end if
end while
Output: Trained policy network Q

Hyperparameter search

In terms of the hyperparameter search for the Rainbow DQN several hyperparameter tuning configurations have been tried. Among the most important looked for on the hyperparameter search are both the discount factor and Learning Rate. The optimal values for them are the following: Gamma: 0.9, lr = 0.0001

Results and visualizations

To validate our novel exploration logic implementation with *dynamic noise injection on Noisy Layers* when a mean reward plateau is found we compared the training performance of this implementation with another established **Exploration-Exploitation** trade-off technique: **Epsilon-greedy**.

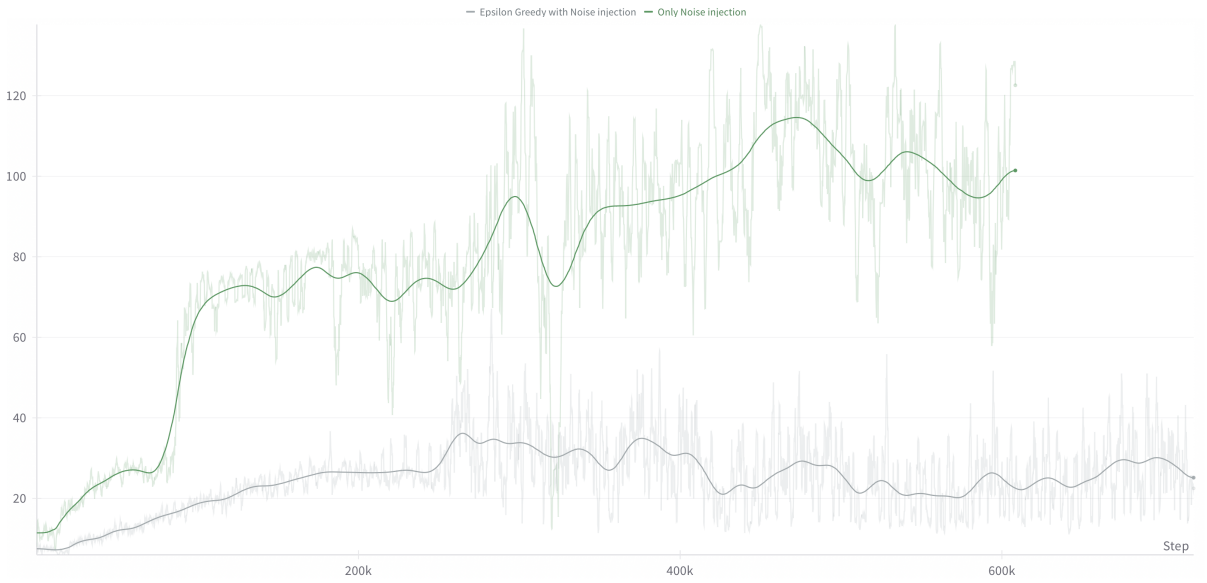


Fig. 4: Exploration vs Exploitation techniques comparison

It must be stated that other than the Exploration-Exploitation technique all the other components for the experiment have remained constant in order for the comparison to be fair. Therefore, the model architecture, dimension, and hyperparameters are the same.
As shown in Figure 4, our method demonstrates a ***significant improvement in mean reward over time*** when encountering a ***learning plateau***, whereas the ***epsilon-greedy technique*** does not. We argue that this phenomenon occurs because our method avoids taking completely random actions during exploration phases.
Note that the ***Plateau-Aware Noise Injection*** applied to the Noisy Layers ensures that exploration is *not completely random*. Specifically, the perturbation (*Noise Injection*) caused by the ***Standard Gaussian Noise*** ($\mathcal{N}(\mu, \sigma)$) does not fully disrupt the ***previously learned weights*** of these layers. Therefore, we argue the resulting exploration ***action distribution*** better resembles a ***Probabilistic Gaussian Distribution*** of the possible Q-values, rather than relying on ***random action selection***.

Figure 5 helps us visualize the actual ***change in strategy*** within the learned policy due to the ***Plateau-Aware Noise Injection*** training technique. By analyzing the plots at approximately the $100k$**-step mark**, the action selection distribution becomes ***more evenly spread***, while still maintaining a preference for the ***up action***. This action selection strategy makes ***semantic sense*** when considering the dynamics of the *Frogger environment*. Notably, as seen in Figure 4, the ***Rainbow training*** exhibits a clear improvement at the $100k$-step mark, further validating the ***enhanced policy action selection strategy***.

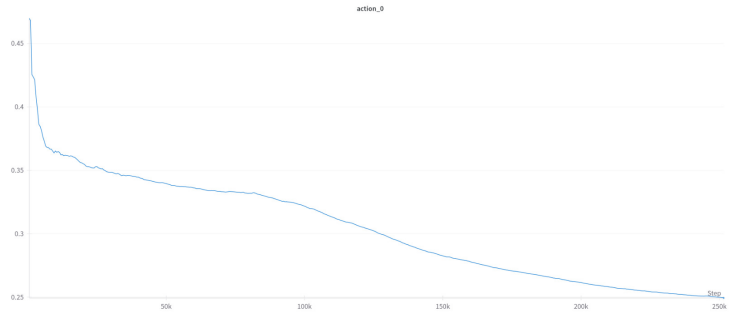
It is also worth mentioning that the injected noise in the distribution does not deviate beyond 5% of the original Gaussian Distribution's standard deviation (σ_0). By employing this technique and introducing an ***eventual decay of Noise Injection*** during the exploration phase, we ensure that exploration ***gradually decreases over time***. This gradual decay allows the agent to enter into subsequent ***exploration phases*** of training, effectively balancing exploration and exploitation.

1.5 Global Results of Part 1

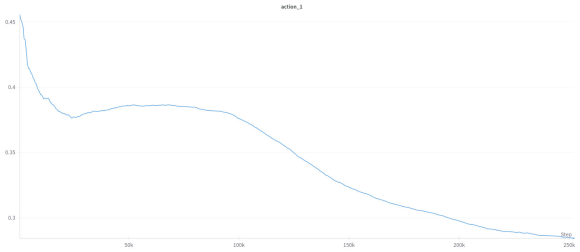
To establish the best algorithm for the first part of the project we have compared the best experiment of each of the algorithms.

The comparison can be seen in Figure 6, we see that our rainbow approach is capable of learning much faster at the first 200.000 episodes but for longer trainings our simple method outperforms it.

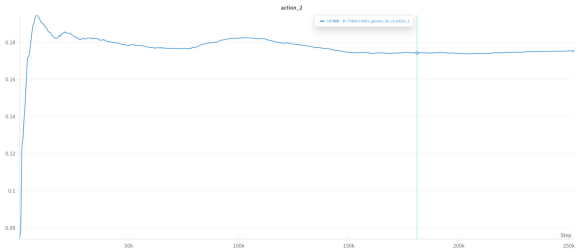
For this reason and the conclusions extracted from the hyper parameter search, we consider that our rainbow approach could outperform the simpler DQN algorithm, if a further hyper parameter search was performed. In this parameters we would require than our model would be able to explore more on further steps of the training.



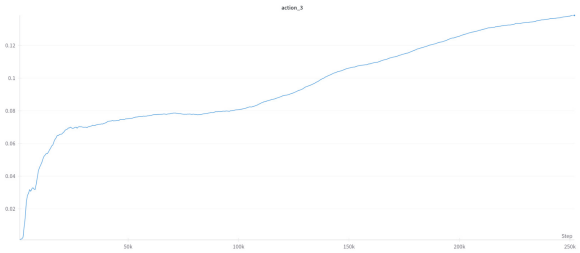
(a) Action 0: No-op



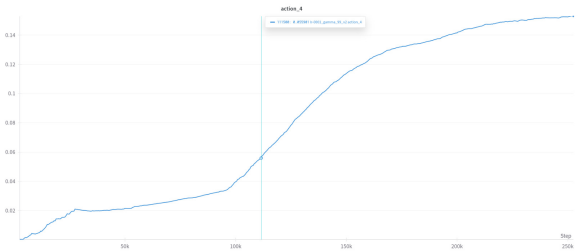
(b) Action 1: Up



(c) Action 2: Right



(d) Action 3: Left



(e) Action 4: Down

Fig. 5: Action selection evolution throughout Rainbow DQN training.

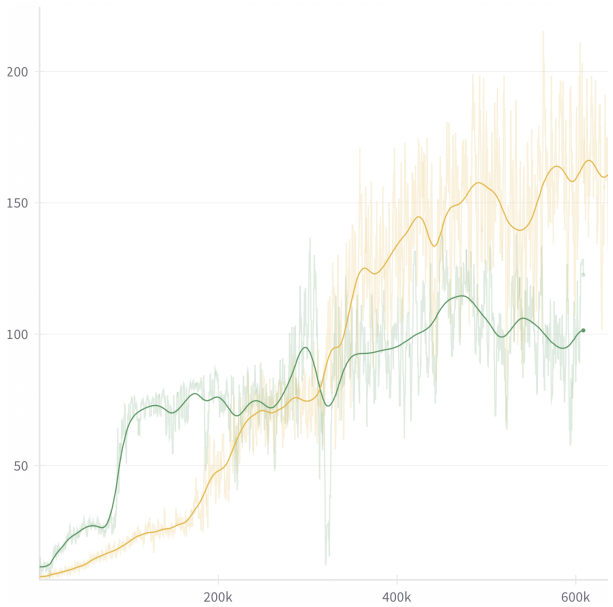


Fig. 6: Comparison of the rolling average reward, 10 episodes, during training of our simple and Rainbow DQN approaches, with the best parameters found.

One of the most important points to take into account in the conclusions of this **Rainbow DQN** algorithm is the fact that the ***percentage of noise distribution injected in the Noisy Layers*** plays a critical role in the ***Exploration vs. Exploitation trade-off***.

Due to the scale by which the network has to learn the ***Mean and Standard Deviation of the Initial Noisy Layers***, the indeterminism of this optimization problem (*finding the optimal values of the weights*) strongly encourages exploration during the ***initial stages***. At the same time, the ***Plateau-aware Noise Injection*** exploration technique helps mitigate the ***local minima problem***. Results suggest that the scale of noise injected during these ***dynamic new exploration phases*** should be higher.

By checking the ***Standard Deviation of the evaluation results***, we can observe that the network has converged to an ***optimal policy*** with little incentive for further exploration. This can be clearly seen if we compare at the distribution of rewards obtained by the Rainbow Model and the DQN with some extensions. This comparison can be seen at Figures 7 and 8.

Fig. 7: Rewards over the 100 test episodes for the DQN with some extensions, in which the red line corresponds to the maximum reward and the green one to the mean

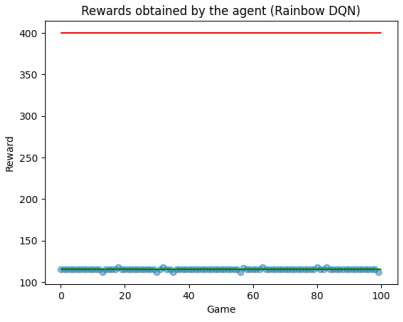


Fig. 8: Rewards over the 100 test episodes for the Rainbow DQN, in which the red line corresponds to the maximum reward and the green one to the mean.

Considering all this information, we believe that at this moment our best method is the ***simpler implementation of DQN***. However, with further ***parameter tuning***, our ***Rainbow approach*** could become the ***optimal solution***.

2 Ms. Pac-Man ALE environment

2.1 Description of the environment

For this second part of the project, we have used Ms. Pacman, which is a spin-off sequel from Pac-Man. In this game, the player, which controls Ms. Pac-Man, is tasked to eat the pellets present at a maze, at the same time in which it avoids being eaten by the ghosts present at the maze. When Ms. Pacman eats a special pellet, the power pellets, the situation changes for a short period of time in which she can eat the ghosts it encounters. An illustration of the game can be seen in Figure 9.

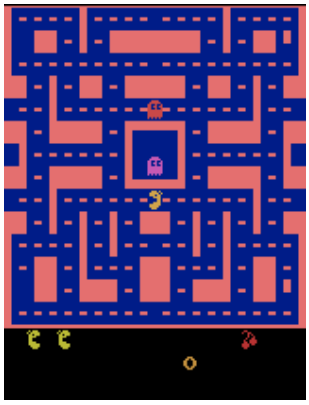


Fig. 9: An image of the initial state of the Ms. Pac-Man game.

Note that the actions that the agent can take in the environment are discrete, specifically 8. Below, is the table of action indices and meanings:

- **0**: NOOP
- **1**: UP
- **2**: RIGHT
- **3**: LEFT
- **4**: DOWN
- **5**: UPRIGHT
- **6**: UPLEFT
- **7**: DOWNRIGHT
- **8**: DOWNLEFT

2.2 Prepossessing

For both learning approaches we have used the same prepossessing procedure of the images generated by the game. The processing consists of the five standard steps on stable baselines. The first step is Noop reset which obtains an initial state by taking a random number of no-ops on reset. Afterwards the frame skipping is performed in which set a value of 4. We also use a wrapper that when a life is lost gives a simulates as if the game has ended, it does not perform a reset so the following game starts from the final state f the game in which a live was lost. Lastly we resize the images to 84x84, and convert to grayscale. We also clip the rewards given by the game to [-1, 0, 1], depending of the sign. Lastly, we stack the last four observations, in order that the model can understand movement. The images are scaled inside the algorithm used for training.

2.3 PPO

Introduction

As a first model choice in Part 2 we decided to use the **Proximal Policy Optimization** algorithm. The **PPO** is classified as a **Policy Gradient** method, which directly optimizes the policy $\pi_\theta(a|s)$ by adjusting its parameters through gradient ascent to maximize expected rewards.

What sets PPO apart is its emphasis on *small policy update steps* to ensure stable training. Specifically, it introduces a *clipping mechanism* to constrain the size of updates to the policy, preventing drastic changes that could destabilize learning. If the step size is too large, the policy may move far from the optimal direction, potentially falling into a suboptimal region with limited chances of recovery. Conversely, overly small steps slow down training and reduce overall efficiency. By balancing these updates, PPO achieves a good trade-off between exploration, stability, and learning speed.

Hyperparameter search

To perform the parameter search we have decided to start from an already known set of parameters that worked well for PPO. For this reason, we have used, as a starting point, the parameters that can be seen at the GitHub repository for RL Baselines3 Zoo, which are optimized for PPO. The entire set of parameters can be seen at the GitHub repository.

From this parameters we have explored the effect of the learning rate and discount factor values.

We first decided which was the best discount factor for the combination of this game and PPO. For this task we tested three different possibilities (0.9, 0.95 and 0.99). At Figure 10, we see that in this case the higher value of discount factor is clearly worse during the entire training episodes. Between 0.9 and 0.95 the differences are smaller, being close to each other during the training procedure but the experiment using a 0.9 discount factor differentiates at the last 2.000.000 frames showing a better performance. This is shown at the average reward being over 500 units greater.

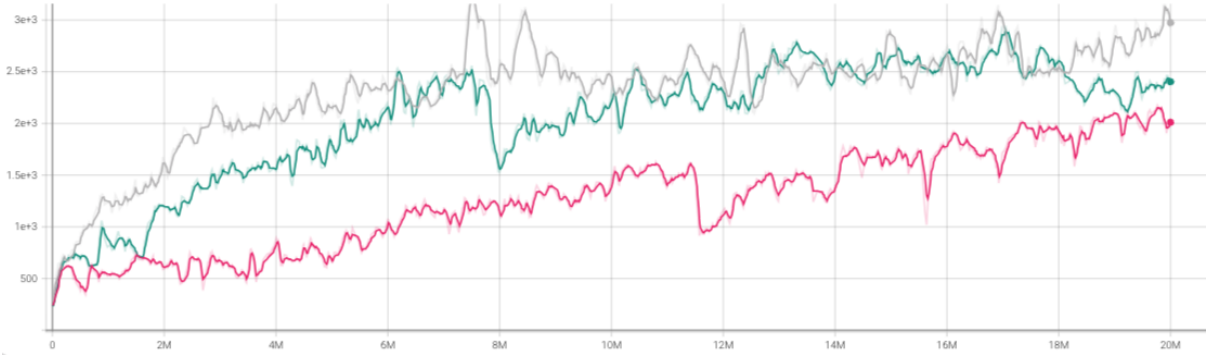


Fig. 10: Comparison of the reward evolution during training for different values of discount factor for PPO model in Ms. Pac-Man game. The gray experiment corresponds to a discount factor of 0.9, the green one to 0.95 and the pink one to 0.99. The three experiments use the same learning rate of 0.00025

After determining the best discount factor we experimented with different learning rates, Figure 11, in this case we do not see huge differences between them. Which translates into a lower importance of learning rate in comparison to discount factor, for this combination of model and game. Even though we can see that the best performance is achieved with a learning rate of 0.00025, which is the same as the optimized value from RL Baselines3 Zoo.

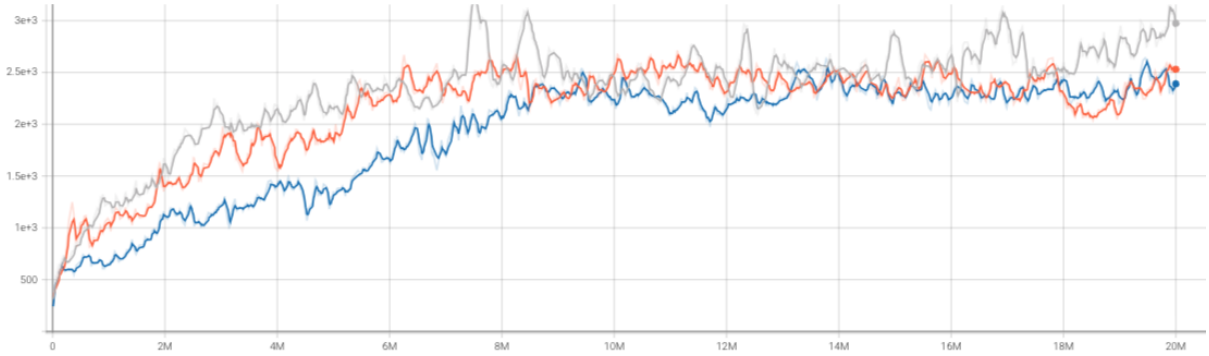


Fig. 11: Comparison of the reward evolution during training for different values of learning rate for PPO model in Ms. Pac-Man game. The gray experiment corresponds to a learning rate of 0.00025, the orange one to 0.00075 and the dark blue one to 0.000075. The three experiments use the same discount factor of 0.90

Results and visualizations

If we evaluate the model with the best parameters, we see an average reward of 4403.3 with a standard deviation of 2190.21 over 100 episodes. This means that usually our policy is completing the first screen of Ms. Pac-Man, and advancing a significant part on the second one. But, there are also cases in which losses all the lives quickly.

The model behavior can be seen at the video generated form an episode of this model, which is available at our public GitHub repository. If we take a look at a video of an episode we see that the model has learned to navigate well through the maze. It is important to notice that in the case of this best model we see that the model is not only following a learned path through the maze, it takes into account the positions of the ghosts avoiding, or eating, them depending on the situation.

2.4 A2C

Introduction

As for the second model choice in Part 2, we have decided to utilize an **Advantage Actor Critic** technique. The **Advantage Actor-Critic (A2C)** algorithm combines the strengths of two methods: policy-based methods (the actor) and value-based methods (the critic).

The **actor** is responsible for learning the policy $\pi_\theta(a|s)$, which determines the probability of taking an action a given a state s. It updates the policy parameters using gradients weighted by the *advantage* $A(s,a) = Q(s,a) - V(s)$, which measures how much better an action a is compared to the average action in that state.

Note that the **critic** estimates the *value function* $V(s)$, which represents the expected cumulative reward from state s following the current policy. Here, the critic's role is to evaluate how good a state is, helping the actor make better decisions by providing a baseline for comparison.

We must alse point out that the advantage function $A(s,a)$ reduces variance in policy updates by subtracting the baseline $V(s)$ from the expected return $Q(s,a)$. This comparison theoretically ensures the policy focuses on actions that outperform the average, leading to more stable and efficient learning.

Hyperparameter search

To perform the parameter search we have decided to start from an already known set of parameters that worked well for A2C. As before we have used the GitHub repository for RL Baselines3 Zoo, in which there are the set of parameters of A2C optimized for atari games. The entire set of parameters can be seen at the GitHub repository.

As before, we have explored the the effect of the learning rate and discount factor values. We started experimenting with the discount factor. We evaluated two different values, 0.9 and 0.99, we see that

the lower discount factor allows the model to learn quicker. For this reason, even though there are not noticeable differences at the final reward, we decided to use a discount factor of 0.9. This comparison can be seen in Figure 12.

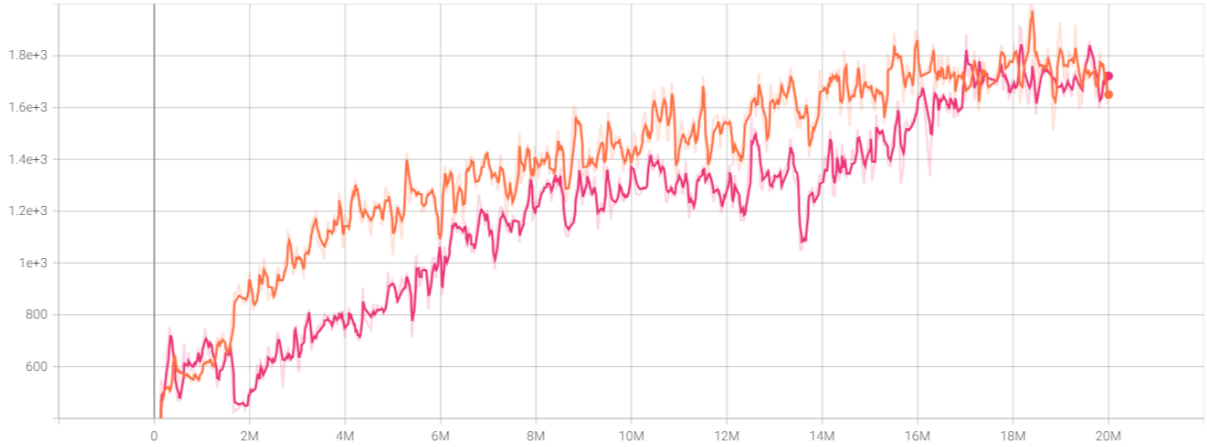


Fig. 12: Comparison of the reward evolution during training for different values of discount factor for A2C model in Ms. Pac-Man game. The orange experiment corresponds to a discount factor of 0.9 and the pink one to 0.99. The two experiments use the same learning rate of 0.00025

Once we selected the discount factor, we experimented with the effect of the learning rate. We have tried the following values: 0.007, 0.001, 0.00025 and 0.00005 without any scheduler to reduce the value as the training progressed. Also, we tried using a value of 0.0015 with an step scheduler.

Out of this experiments we see that the highest and lowest values do not allow training correctly. Within the rest of experiments we do not see any noticeable difference except at the final part of training, in which the experiment with a learning rate 0.001 clearly outperforms any other setup.

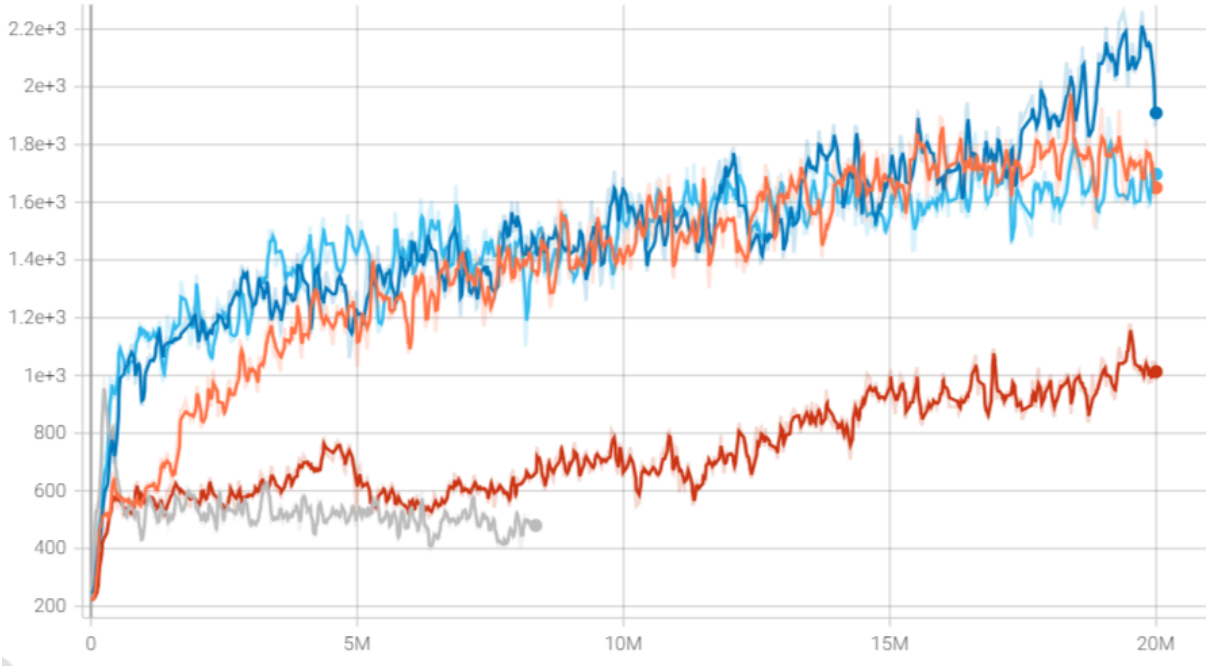


Fig. 13: Comparison of the reward evolution during training for different values of learning rate for A2C model in Ms. Pac-Man game. The dark blue corresponds experiment corresponds to a learning rate of 0.001, the sky blue to 0.0015 with an scheduler to reduce it during training, orange to 0.00025, red to 0.00005 and finally gray to 0.007. All the experiments use the same discount factor of 0.9.

Results and visualizations

If we evaluate the model with the best parameters, we see an average reward of 2146.5 with a standard deviation of 785.015 over 100 episodes. This means that our policy sometimes is completing the first screen of Ms. Pac-Man, with a small journey on the second one.
The model behavior can be seen at the video generated form an episode of this model, which is available at our public GitHub repository. If we take a look at a video of an episode we see that the model has learned a path through the maze. This path allows the model to navigate through this first screen but it can usually crash into a ghost.

2.5 Global Results of Part 2

If we compare the results we see that our model using PPO clearly outperforms the A2C one. It has almost the double average reward, which is also translated into a much the behavior.
We think that this might be caused due to the better stability of the PPO algorithm, which it is allowing the model to explore better.
We argue that training dynamics of the Advantage Actor-Critic have led to a worse policy than the Proximal Policy Optimization method due to the overestimation of the environmental dynamics and the action.

3 Pong World Tournament

In part's 3 Pong, you control the right paddle, you compete against the left paddle controlled by the computer. You each try to keep deflecting the ball away from your goal and into your opponent's goal.

Note that the actions that the agent can take in the environment are discrete, specifically 5. Below, that is the table of action indices and meanings:

- **0**: NOOP

- **1**: FIRE
- **2**: RIGHT
- **3**: LEFT
- **4**: RIGHTFIRE
- **5**: LEFTFIRE

3.1 Learning algorithm used

Because of the great performance we obtained using PPO at the second section, we decided to stick to this algorithm for the last part of this project. In which a stable and great performing algorithm is useful due to the methodology we have used.

3.2 Methodology

In order to train our agents for Pong, we have decided to use an adversarial method. In this method each of the models is trained to beat the other, in this way as training advances both agents need to improve and find new strategies to be able to keep scoring points. Also, it provides the advantage that the agents can continue improving, even after becoming good players, as they demand more from each other each time. This would not happen if we trained our agent versus an already trained agent, as it would arrive to one point in which it would always win, not being able to improve.

In more detail of our implementation, we have decided to train one agent at the time to make training more stable. So, our method works by training for a predetermined number of steps one of the agents with the other frozen, once the specified steps have been made they interchange positions (Frozen agent now becomes the one being trained and vice versa). The details of this training technique can be seen in algorithm 2.

Algorithm 2 Adversarial Training with Role Switching for PPO Agents

Input: Two PPO agents A_0 and A_1 with policies π_0 and π_1, adversarial iterations n_{iter}, training timesteps per iteration T_{adv}
Initialize: Policies π_0 and π_1, alternating iteration index $i \leftarrow 0$
while $i < n_{\text{iter}}$ **do**
  if $i \mod 2 == 0$ **then** ▷ Train A_0 (Online) against A_1 (Target)
    Fix the target policy π_1 for agent A_1
    Train the online policy π_0 for T_{adv} timesteps:

$$\pi_0 \leftarrow \arg\max_{\theta_0} \mathbb{E}_{\tau \sim \pi_0, \pi_1} \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

    Update agent A_0's policy with the learned parameters
  else ▷ Train A_1 (Online) against A_0 (Target)
    Fix the target policy π_0 for agent A_0
    Train the online policy π_1 for T_{adv} timesteps:

$$\pi_1 \leftarrow \arg\max_{\theta_1} \mathbb{E}_{\tau \sim \pi_1, \pi_0} \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

    Update agent A_1's policy with the learned parameters
  end if
  $i \leftarrow i + 1$ ▷ Increment adversarial iteration counter
end while
Output: Final trained policies π_0 and π_1

Once we defined set up we needed to perform the search of hyper parameters, that worked better at this game for PPO.

3.3 Parameter search

To perform the hyperparameter search we have used ALE Pong environment, because it provided us with an already trained opponent. As in Section 2, we have decided to start the hyperparameter search with the optimized parameters for PPO at Atari games from RL Baselines3 Zoo. We have explored the effect of different learning rate and discount factor values.

In the experimentation on discount factor values we have used values of 0.9, 0.95 and 0.99. Maintaining the rest of hyperparameters equal between the different experiments. The reward evolution during training can be seen in Figure 14. In this figure is seen that as the value of the discount factor becomes smaller, PPO works worse for this game. Being 0.99 the best value out of the ones tried.

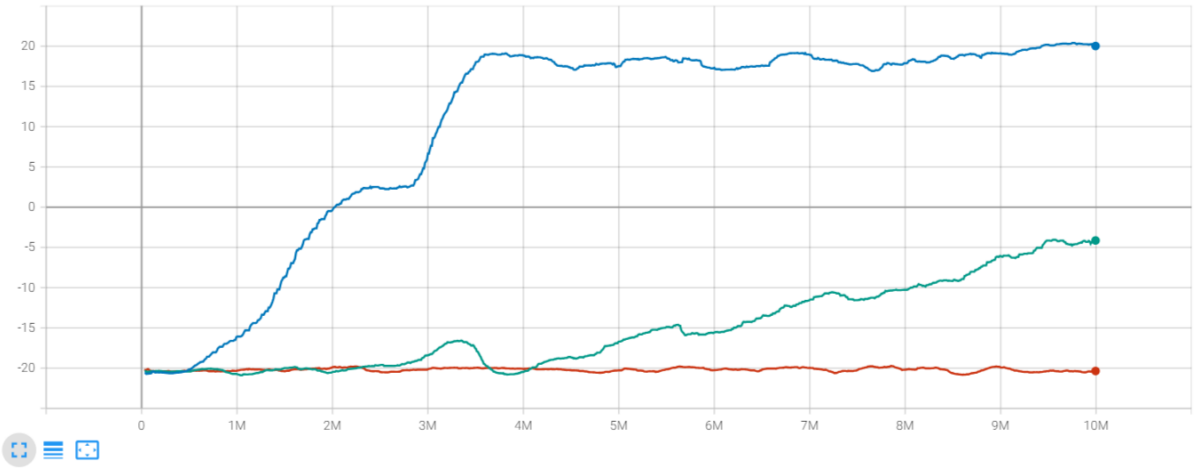


Fig. 14: Evolution of rewards obtained during training for different discount factors. Blue corresponds to 0.99, green to 0.95, and red to 0.9.

Once we determined 0.99 as the best discount factor for Pong, in combination with PPO, we experimented with the learning rate values. In this case, we tested the following learning rate values, 0.00075, 0.00025 and 0.000075. The evolution of rewards during training is seen in Figure 15, in which it is seen that the intermediate value is clearly better. The higher value produces more instable training, while the smaller one produces a very slow learning, increasing the probabilities of getting stuck on a local minimum.

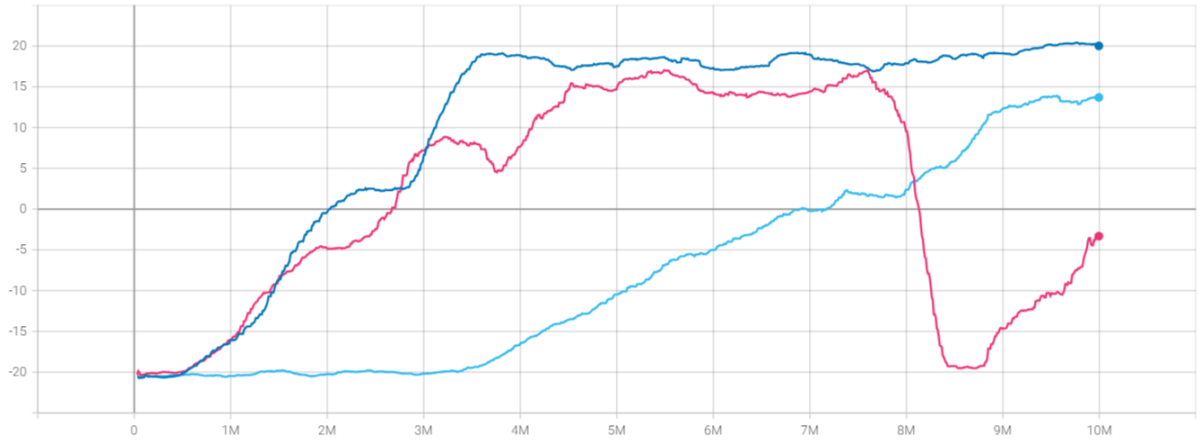


Fig. 15: Evolution of rewards obtained during training for different learning rates. Dark blue corresponds to 0.00025, pink to 0.00075, and sky blue to 0.000075.

After this exploration of parameters, we established that the best parameters were a discount factor of 0.99 and a learning rate of 0.00025, in combination with the rest of parameters from RL Baselines3 Zoo.

3.4 Model Training

Once we determined the best hyperparameters for the learning algorithm and game combination, we started with our method to train the right and left agents for Pong.
We decided to start the training process from zero in our agents, in this way at the start both agents would play random. In this way, the models could learn slowly together, and none of the agents would become much better than the other. For the same reason, we decided to perform 100.000 training steps for each agent in each of the adversarial cycles. We performed 640 iterations, which translates into each agent being trained for 32.000.000 frames in total.

During the training of the model we saw a gradual improvement on the capabilities to play Pong. As the training progressed, we saw how both agents started implementing more complex strategies, even learning to produce effects on the ball, making it more difficult to predict where the ball was going for the other agent. Another thing it can be noticed is that as more training steps were performed, the agents tried to play more with the upper or lower flat areas of the rackets, which can increase the speed of the ball.

If we try our right agent performance versus the default player of Pong, we can see an average reward of -4.6 over 100 episodes, with an standard deviation of 5.64. And we can see that our agent is playing well versus the default opponent, but is not capable of winning always. This can be caused as during the last training steps the ball was thrown in a non usual way, most of the times, by our other policy. We consider that training our adversarial models for more iterations would be beneficial for the policy generalization, as during the last 200 adversarial iterations the average reward on the default pong improved by five units.

In comparison, the best policy trained versus the the default opponent obtained a mean reward of 19.4 with an standard deviation of 1.36. This can be explained as this agent was trained to exploit the deficiencies of the default agent, focusing on throwing the ball in the ways in which the agent cannot reach it. Our agent instead has learned a more general policy, which we expect to work on versus more opponents.

3.5 Results

The last part of our multi agent Pong methodology is to evaluate the multi agent performance of the trained agents. If we take a look at the video from a game generated in the Petting Zoo Pong environment we can notice that our agents have learned to play, generating long games. Also, during this game is noticed that they try, and accomplish, to generate effects and increase speed of the ball making it more difficult for the other agent to respond. Lastly, we consider that this agents seem to be able to generalize to a broader range of situations, than our model trained on the default environment, due to the dynamic environment in which they were trained.

If we take a look at the results over 100 matches between this agents we see the left one achieves a 90% win rate, and has accomplished 20.5 points on average. On the other side the average punctuation was 14.92. This is caused as on our method there will always be an agents that has been trained one more time than the other, which makes the model able to win much more.

4 Repository

Al the code developed for this project can be seen at https://github.com/joanlafuente/Project-Paradigms.

References

1. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–533. https://doi.org/10.1038/nature14236
2. Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 32, No. 1). https://doi.org/10.1609/aaai.v32i1.11796
3. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*. https://doi.org/10.48550/arXiv.1707.06347
4. Sutton, R. S., Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.
5. Bellemare, M. G., Naddaf, Y., Veness, J., Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, 253–279. https://doi.org/10.1613/jair.3912
6. Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., ... Legg, S. (2018). Noisy networks for exploration. In *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.1706.10295
7. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*. Available at: GitHub Repository