PARADIGMS OF MACHINE LEARNING - PROJECT REPORT

Pere Mayol Carbonell, NIU: 1669503
Andreu Gascón Marzo, NIU: 1670919
Joan Bayona Corbalán, NIU: 1667446

ÍNDEX

1. KABOOM!

Kaboom! is a classic Atari 2600 game in which the player must use buckets to catch bombs dropped by a "Mad Bomber." This seemingly simple task presents a challenging environment for reinforcement learning (RL) agents due to its dynamic nature and increasing difficulty.

1.1 Game Dynamics

- **Objective:** Catch falling bombs to score points and avoid losing all your buckets.
- **Controls:** The player controls the horizontal movement of buckets using a paddle controller.
- **Challenges and Dynamics:**
  - **Increasing Speed:** Bombs fall progressively faster as the game progresses (depending on the level).
  - **Limited Buckets:** The player has a limited number of buckets (three). Missing a bomb results in the loss of a bucket and return to the previous level (where
  - **Game Over:** The game ends when all buckets are lost.
  - **Reward System:** Points are awarded for each bomb caught and each bomb has a point value associated with it. This value depends on how fast the bomb is falling. Faster bombs are worth more points. Bombs are organized into 8 groups. Each group has a different number of bombs and a different point value per bomb as shown in the table:

# of Bomb Group	# of Bombs in Group	Point Value of Each Bmb	Point Value of Group	Cumulative Score
1	10	1	10	10
2	20	2	40	50
3	30	3	90	140
4	40	4	160	300
5	50	5	250	550
6	75	6	450	1000
7	100	7	700	1700
8	150	8	1200	2900

1.2 Challenges in the Reward System

- **Infrequent Feedback:** Unlike games with continuous feedback, Kaboom! Only provides rewards when a bomb is caught. This makes it hard for the agent to learn which actions were good and which were bad.
- **Long-Term Planning:** Catching bombs in later, faster groups yields more points. However, the agent must first learn to survive the early stages and progress through the game. This requires long-term planning and prioritizing actions that lead to future rewards. Most importantly, a lot of training hours to be able to arrive in elder states.
- **Adapting to Change:** The game's increasing difficulty requires the agent to continuously adapt its strategy. What works in the early stages might not work later when bombs fall faster.

1.3 Chosen Algorithms and Solutions

For our environment, we decided to use three methods, a Vanilla DQN, Double DQN and an Enhanced Reward System.
For all models except for the last one, we used the same wrappers to process the observations.

- ➔ **EnvCompatibility**: Ensures compatibility between different Gymnasium versions by standardizing the output of step() and reset().
- ➔ **MaxAndSkipEnv**: Combines consecutive frames and skips frames to speed up gameplay and reduce flickering. We chose 4 skip frames.
- ➔ **ProcessFrame84**: Resizes, crops, and converts frames to grayscale for simpler and more efficient processing.
- ➔ **BufferWrapper**: Stacks multiple frames together to provide temporal information to the agent.
- ➔ **ScaledFloatFrame**: Normalizes pixel values to a range of 0.0 to 1.0 for better performance in machine learning.
- ➔ **EnhancedRewardWrapper**: Starts the game with the "FIRE" action. (this wrapper will be modified with the third method)
- ➔ **RecordVideo**: Records videos of the agent's gameplay at specified intervals.

1.3.1 Basic DQN (Deep Q-Network)

- ➔ **Random Experience Replay:** Sampling random batches helps break the correlation between consecutive experiences, making the learning process more stable.
- ➔ **Target Network:** A target network is introduced as a separate copy of the main network. This target network is used to estimate the Q-values for the next state. The target network is updated less frequently than the main network.
- ➔ **Epsilon Decay:** Epsilon decay gradually reduces the value of epsilon over time. This means that the agent starts with a high exploration rate (more random actions) and gradually shifts towards more exploitation (choosing the best actions based on its learned Q-values)

1.3.2 Modified DQN

The chosen modifications to the standard DQN architecture are not arbitrary. Given our environment's sparse and delayed reward nature, these are key modifications that will help the agent understand the long-term consequences of its actions and learn to prioritize reaching later stages with higher-value bombs.

1.3.2.1 Prioritized Experience Replay

Prioritized Experience Replay (PER) stores agent experiences with priority values based on their significance (like unexpected events). It then samples these experiences non-uniformly, giving **preference** to those with higher priority, allowing the agent to learn more effectively from important events. To avoid bias from oversampling important events, PER uses **importance sampling weights** to correct for non-uniform sampling. This ensures the agent learns accurately while focusing on the most critical experiences for improved performance.

- ➔ **Break Correlation:** In Kaboom!, consecutive experiences can be highly correlated (e.g., catching a series of slow bombs in the early stages). Sampling from the buffer breaks these correlations, which helps the agent learn a more general strategy and avoid overfitting to specific sequences of falling bombs.
  In this context, a Prioritized experience replay (PER) tweak is expected to work well, since it assigns a priority to each experience based on how "**surprising**" or "**important**" it was. Experiences with **larger errors** in the agent's predictions (meaning the agent learned a lot from them) have a higher priority.

- ➔ **SumTree for Sampling:** SumTree allows time complexity for sampling an experience to be reduced from **O(N) to O(logN)**. That makes a huge improvement in terms of training time, especially for large buffer sizes.
  SumTrees efficiently manages prioritized experience replay (PER) using a **binary tree structure** where leaf nodes store transition priorities, and parent nodes store the **sum** of their children's priorities. Sampling is performed by traversing the tree using a **random value scaled to the total priority sum**, ensuring transitions are sampled proportionally to their priorities. When a priority is updated, the corresponding leaf node is adjusted, and changes propagate upward to keep the parent sums accurate.
  For example: *To sample a batch of size 64 from a SumTree, you divide the total priority sum SSS of the tree into 64 equal ranges, each of size S/64. For each range, a random value is sampled uniformly within the range, and the SumTree is traversed to find the corresponding leaf node (transition).*

1.3.2.2 Double DQN

Standard DQN can sometimes **overestimate** the value of actions. This happens because it uses the same network to both select the best action and estimate its value. Accurate value estimation is crucial for making informed decisions in Kaboom!, where rewards are sparse and delayed.

Double DQN addresses the overestimation bias in standard DQN by **decoupling** action selection and value estimation. The online network selects the action with the highest Q-value, while the target network evaluates the Q-value for that action. This separation results in more stable and accurate value updates, improving learning performance.

1.3.3 Enhanced Reward System

One of the main challenges in level-based games, where each level becomes increasingly difficult, is successfully reaching the final stages. Techniques like Transfer Learning (leveraging pre-trained models) or Curriculum Learning (gradually increasing the difficulty) could have been helpful, but these were not feasible due to two key limitations:

- ➔ **Lack of Information:** There was very little data available about the game online.
- ➔ **Limited ALE/Gymnasium Feedback:** Critical information, such as the current level, lives (buckets) lost, or the bucket's position at a specific time, was not adequately provided.

Despite these challenges, learning was achievable by using well-tuned hyperparameters. However, to further accelerate the learning process, we explored the use of intermediate rewards.

1.3.3.1 RAM approach

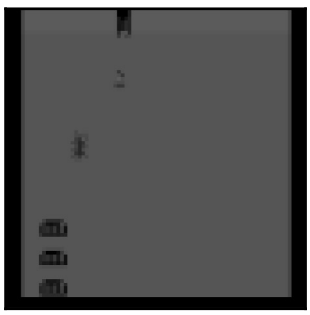
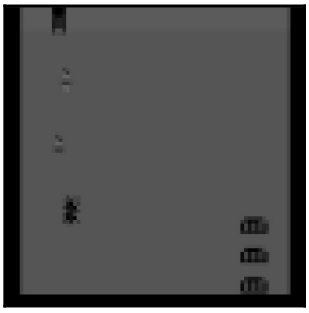
The Atari 2600 was developed with just 128 bytes of RAM, and the ALE/Gymnasium library provides a method to access these bytes. These bytes contain detailed information about the game's state, such as the bucket's position, bomb positions, level count, bomb speed, and scores.
Our objective was to use this information to calculate intermediate rewards based on specific criteria —for example, penalizing for missing a bucket (losing a life) when a bomb explodes, or rewarding based on the bucket's distance to incoming bombs.
However, this approach was not feasible for Kaboom! Because:

- ➔ **Lack of Documentation:** The meanings of the RAM array values for this game are not well-documented.
- ➔ **Game-Specific Storage:** Each Atari game stores its information differently, making it challenging to decode the RAM structure without detailed insights.

1.3.3.2 Computer Vision Approach

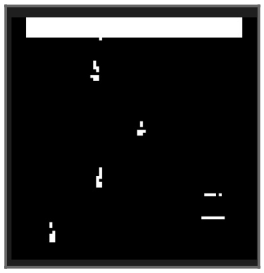
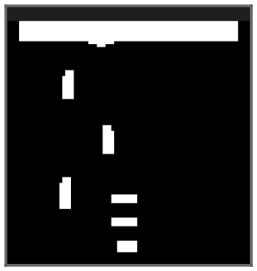
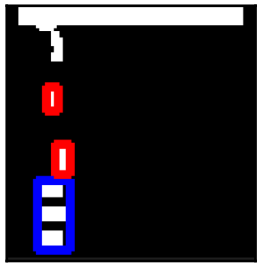
We then opted for a computer vision approach. For that, we created a new wrapper (apart from the preprocessing ones) that, based on the current frame decides whether to give a positive or negative penalty to the agent. The wrapper class *EnhancedRewarWrapper* focuses on two things:

- ➔ **Negative reward in case of an explosion:** Each time a bomb explodes, it makes the rest also explode which causes flickering on the screen. If the mean pixels of the screen pass a certain threshold, that means an explosion, so a negative reward was applied to the total reward of the step.
  These are some images where the explosion was detected and bombs are being destroyed:

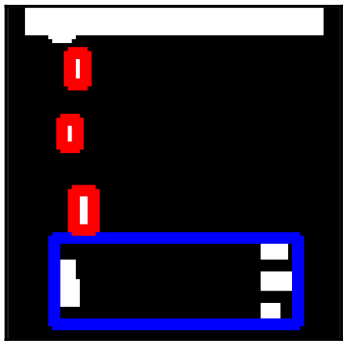



Here we can see how bombs explode.

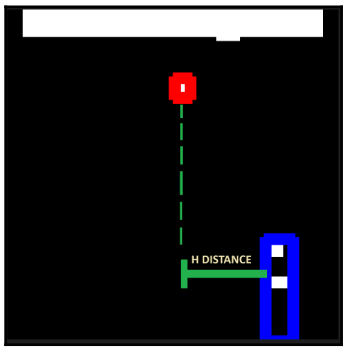
- ➔ **Reward based on the horizontal distance from the buckets to the closest bomb:** To make the agent understand the ultimate goal of the game, we decided to give a positive reward according to the horizontal distance from the buckets to the first detected bomb. For processing the screen we binary thresholded the image and applied dilation to all the white pixels separating the bomb zone with a bigger dilation kernel and smaller for the buckets (in this way bombs are detected better. We then find the contours of bombs and buckets and detect their position in the plane. Finally, in each step, we compute a scaled reward between -1 and 1 according to the current distance of the buckets to the closest bomb.

This process worked more or less as expected although sometimes (rarely) bombs were detected as part of the bucket.



This drawing is the idea of the horizontal distance between the buckets and the bombs.



1.3.4 Reinforce Agent

On top of the models mentioned above, we also tried to apply a **reinforcing agent** to the environment and were ultimately unsuccessful, nevertheless, we wanted to include it in the reward and showcase our trial. We did not quite get why, but the training process seemed to show that the model did not learn anywhere near how we expected it to do, and we ultimately just stayed with the other two models and the rest of the surrounding ideas.

For the training, we wrapped the created environment using several gymnasium wrappers to try and set up a good and useful input for the model to learn. The network and training loop were based on the general ideas of the algorithm taught in class and our versions for the practical activity, evidently with the changes that we thought were necessary.

Ultimately, we were unable to get good results or understand why bad results kept on happening, so we do not think it is appropriate or makes any sense at all to compare this to the functional models explained before. Nevertheless, we thought it was a good idea to add our intentions and what we got in the final result.

1.4 Results and Thoughts

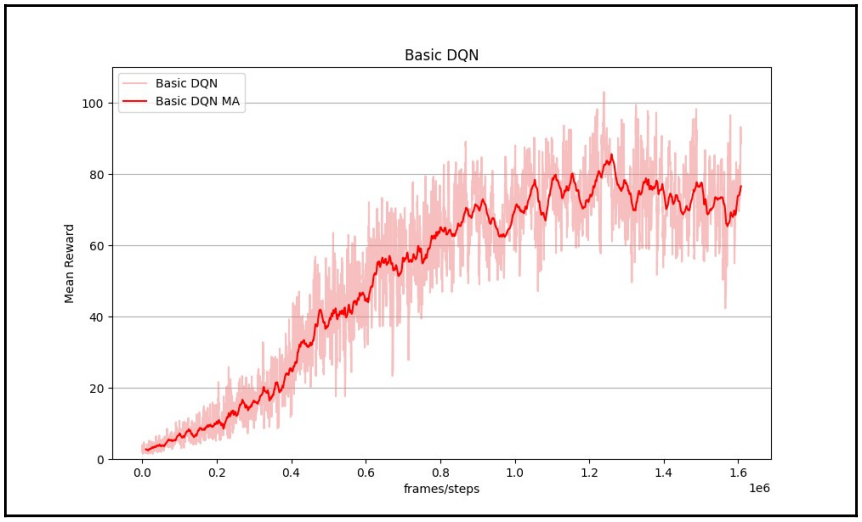
After experimenting with different hyperparameters we found that all 3 methods could learn using the same hyperparameters. After some testing, we decided to train the 3 of them with the following:

- ➔ Learning Rate: 0.0005
- ➔ Epsilon-decay factor: 0.999992
- ➔ 1.6 million steps
- ➔ Discount factor: 0.99

- ➔ Batch Size: 64
- ➔ *bias = 0.4
- ➔ *alpha = 0.6

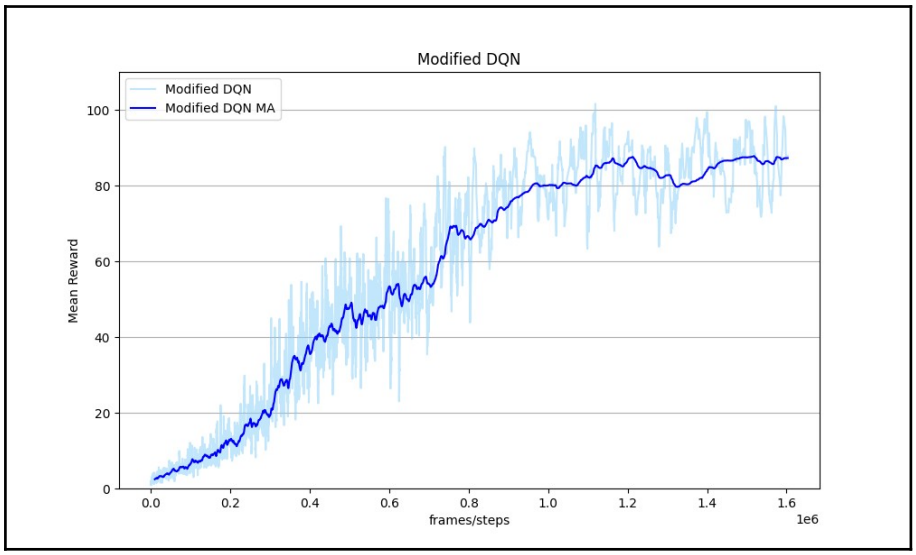
* is for PER

1.4.1 Basic DQN Results



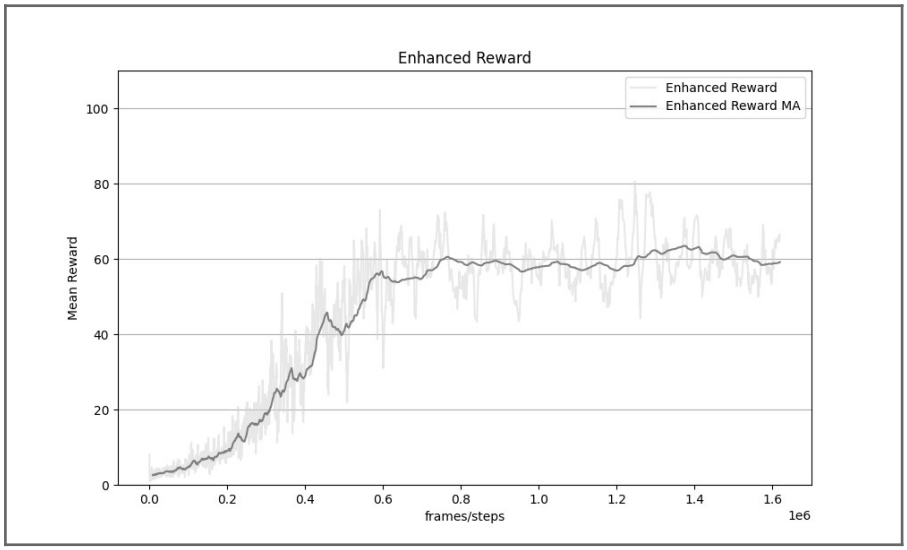
Here we can see the mean reward over episodes in training for the basic DQN method. The model obtains more or less a 70 average reward after 1.6 million steps. The model is quite unstable as seen in the plot.

1.4.2 Modified DQN Results



With DDQN and PER, we obtain much more stable results and it reaches a plateau of around 70 to 80 rewards. This model works as expected by smoothing the curve and accelerating learning.

1.4.3 Enhanced Rewards Result



Here we can see the mean real reward over episodes in training for the Enhanced Reward method. Although we trained the model using the modified reward system, we are using the real reward (obtained directly from the base environment) to compare with other methods.

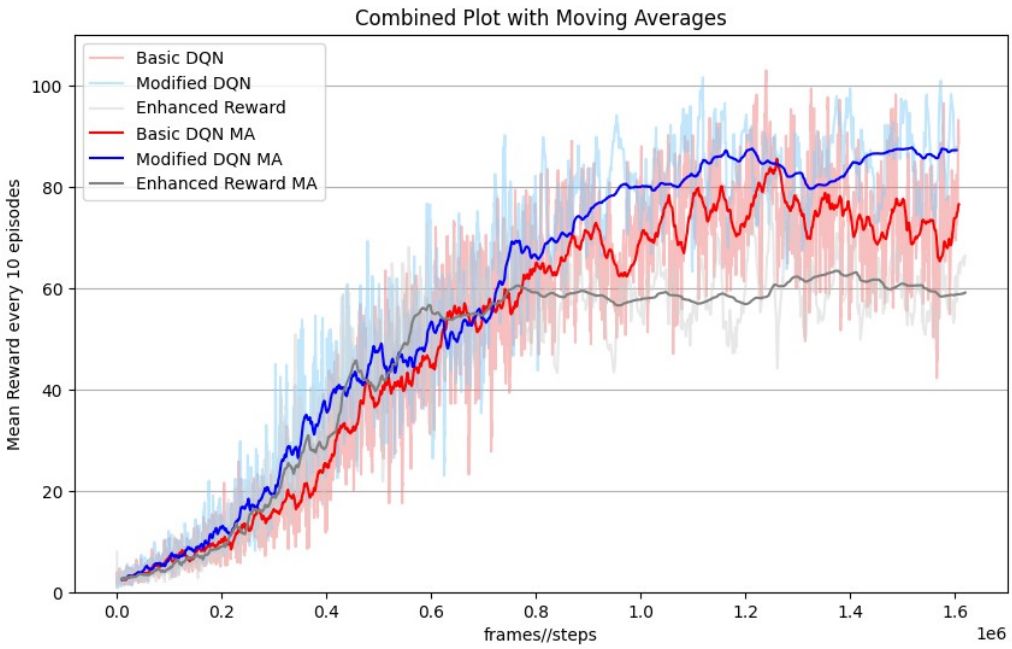
1.4.3.1 Comments on this method

In contrast to the other approaches, this method struggles to progress beyond an average reward of 60. The most likely explanation is that the concept of intermediate rewards was not fully refined. While the execution regarding bomb explosions and coordinated bomb detection was nearly flawless, the reward system's logic may not have been appropriately scaled. For example, the rewards might have been **disproportionately positive,** or because rewards were assigned based solely on **proximity to the nearest bomb**, the agent may have focused exclusively on that bomb while ignoring the others.
It is also worth noting that, although we initially expected the learning time for this model to be significantly longer, it was roughly comparable to the other approaches.

1.4.3.2 Solutions and Improvements

To improve this method, potential solutions include better scaling of the rewards—ensuring they are not overly positive, particularly during the early stages of learning—and assigning a weight to each detected bomb. This weighting could encourage the agent to focus on collecting all bombs rather than fixating on the closest one.

1.5 Testing and Conclusions



This is the comparison of both 3 methods, using the same hyperparameters and the same steps.
As expected the modified version of DQN worked better than the Vanilla DQN, it not only provides more stable learning but achieves slightly better performance in training. On the other hand, as mentioned, the Modified Reward System underperforms clearly both the previous methods.

1.5.1 Testing

We tested all the models over 70 episodes and these are the results:

- ➔ Basic DQN Average Reward over 70 episodes: **56.88**
- ➔ Intermediate Rewards Average Reward over 70 episodes: **13.7**
- ➔ Modified DQN Average Reward over 70 episodes: **80.4**

As seen in training, Modified DQN overpasses the other models by a lot. On the other hand, the intermediate reward version worsened drastically from training obtaining a very slow average.

1.5.2 Conclusion

As mentioned before, the Enhance reward system should be refined in combination with the PER to obtain a better result. These are the main things we should change to obtain better results in training and testing,

- ➔ Assign weights to bomb positions according to the **y-axis** value. Right now the reward about the horizontal distance is only taking into account the **first bomb**.
- ➔ Use a **scheduler** for the **bias** term in PER for more precise bias correction, especially in late states.

➔ Train for **more hours** all the models with a **lower learning rate** and decay factor to try to obtain a better global reward for all models.

2. ASSAULT

Assault is a classic Atari 2600 game in which a mothership drops up to three aliens at a time on the screen. The player's objective is to shoot the aliens whilst avoiding getting hit by the missiles, lasers and fireballs the enemies shoot.

2.1 Game Dynamics

➔ **Objective:** Avoid getting hit by aliens and fireballs coming from the sides. Also hitting the aliens in order to kill them.
➔ **Controls:** The player in this game can move left and right to avoid getting hit, it can also fire right, left and up to end the enemies in the gameplay.
➔ **Challenges and Dynamics:**
- **Enemies variance:** Every time you kill a group of 10 aliens, a different one appears, with small variances, they might shoot faster than the previous ones, some even attack you from the sides, and some of them when you kill them duplicate (become 2 aliens), some can move vertically and not just horizontally.
- **Temperature**: There's a temperature bar in the bottom-right of the screen, the more you shoot, the more it increases, and you could overheat the cannon and die. If you can avoid firing the cannon unnecessarily, then the cannon will cool down.
- **Game over**: The game ends when you reach the maximum score (ie. 999999) or when all canons have been destroyed (you have 3 cannons).
- The environment can be quite random due to the behaviors of aliens, how they appear, where they appear and how they shoot, this complicates considerably training an agent capable of understanding all of these conditions and acting accordingly.

Example of the Assault game

2.2 Chosen algorithms and solutions

For our environment, we decided to use two methods. The first one is the Advantage of the Actor-Critic. The second one is using PPO. They are both based on implementations from the known library stable baselines.

2.2.1 A2C (Advantage Actor-Critic)

2.2.1.1 How does A2C work?

Advantage Actor-Critic (A2C) is a synchronous, deterministic variant of the Actor-Critic family of reinforcement learning algorithms. As the name suggests, A2C consists of two networks:

- **The Actor:** Responsible for deciding what action to take given the current state. It outputs a probability distribution over actions, which forms the policy.
- **The Critic:** Responsible for estimating the value function, specifically the extra reward we get from selecting a given action. This helps guide the Actor by providing feedback on the quality of the chosen actions.

The central concept in A2C is the **advantage function**, which would be like the “extra reward” we get from selecting a certain action. The advantage function is computed as:

$$A(s,a)=Q(s,a)-V(s)$$

If A(s,a) < 0 it means that the chosen action is not good and gives us a value lower than expected in that state. Compared to Actor-Critic (AC), it reduces the variability of policy neural networks and improves the stability of the entire model.

2.2.1.2 Why use A2C?

A2C is popular due to its balance between simplicity and effectiveness. The main reasons for choosing A2C are:

- **Stability:** Using the advantage function reduces the variance of policy gradient updates, leading to smoother learning.
- **Synchronous operation:** Collecting experience synchronously from multiple environments allows for stable and predictable optimization compared to asynchronous approaches like A3C.
- **Efficiency:** A2C is computationally efficient and benefits from vectorized environments, making it faster to train on modern hardware.

2.2.1.3 Solving Assault Using A2C

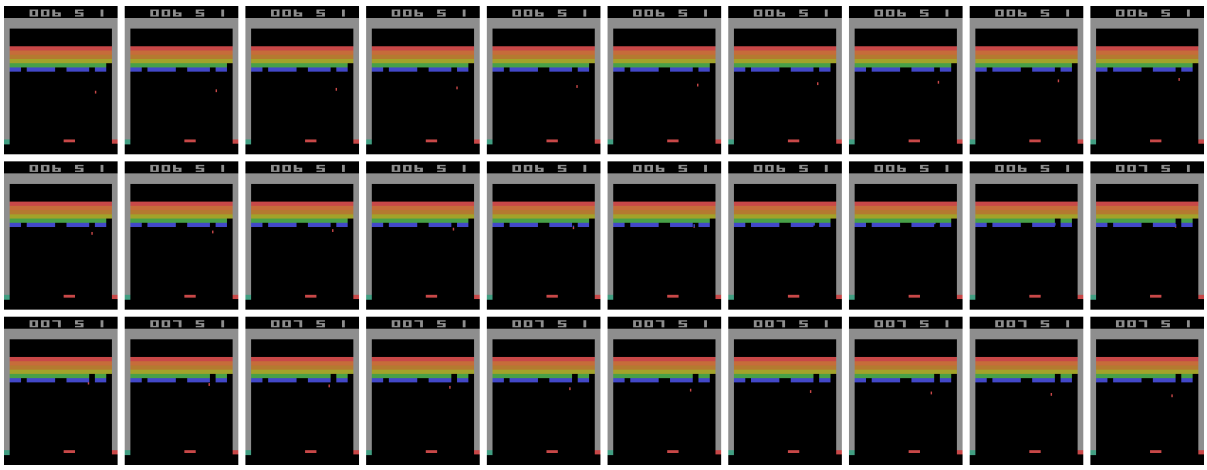
In our case, we used the A2C implementation provided by stable baselines. Our goal was to optimize the training process to achieve stable and effective learning on the "Assault" environment.

Environment Setup:

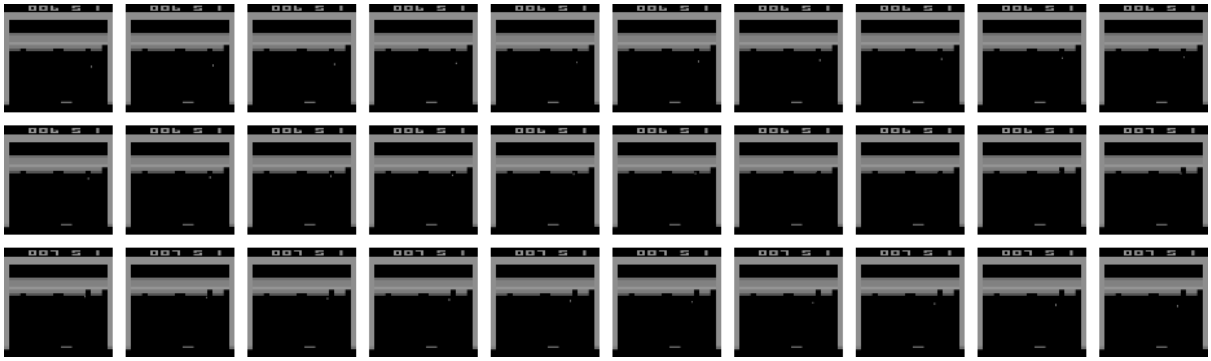
- We used the **make_atari_env,** which includes common preprocessing for Atari games like "Assault" (AssaultNoFrameskip-v4).
- To enhance temporal awareness and decision-making, we stacked four consecutive frames as input using **VecFrameStack**.
- A total of 16 parallel environments were used, ensuring diverse experience collection and faster training due to synchronous interaction across all environments.

The common preprocessing includes of **make_atari_env('AssaultNoFrameskip-v4, n_envs=16, seed=0)** includes:

1. Grayscale Conversion: Converts the RGB Atari frames (210x160x3) to (210x160x1) in order to reduce computation costs.

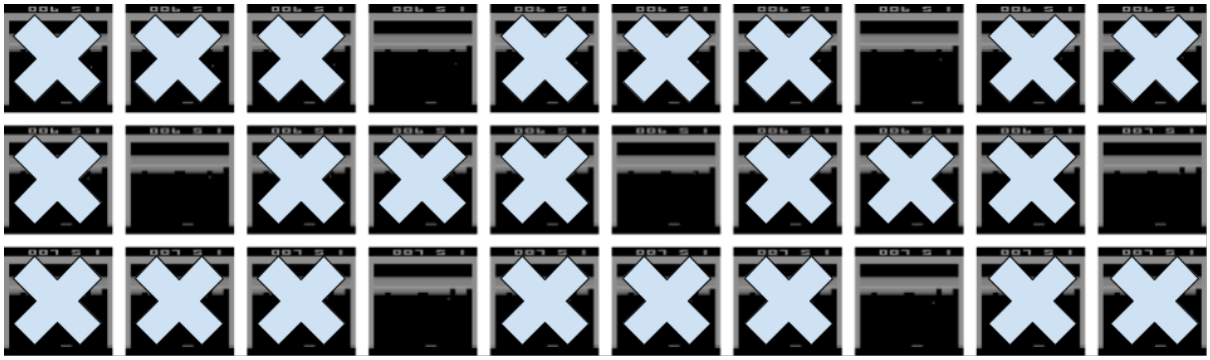


Example of 30 frames in the game of breakout



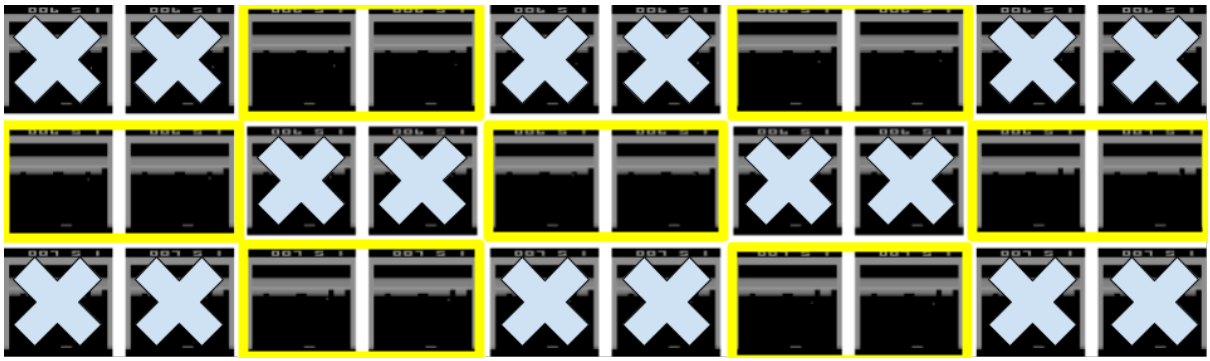
Example of grayscale conversion

2. Frame Skipping: Instead of processing every single frame, the environment skips frames, executing the same action during skipped frames.



Example of frame skipping in the game breakout

3. Max-Pooling Over Two Consecutive Frames: The wrapper takes the maximum pixel values of two consecutive frames to handle rendering inconsistencies.



Two consecutive frames and we take the pixel-by-pixel maximum of the two images

4. Noop Reset: This ensures that the initial state isn't always identical, which prevents the agent from overfitting to deterministic start conditions.

5. Resize to 84x84: Resizes the grayscale frames from (210x160) to (84x84). Reduces the dimensionality of the input, lowering computational costs.

6. Clipping Rewards: Clips the rewards to a range of {-1, 0, 1}. Normalizes rewards to prevent large reward magnitudes from destabilizing the training process.

7. Terminate on Life Loss: Ends the current episode when the agent loses a life, even if the agent has several lives, it makes the agent minimize life loss.

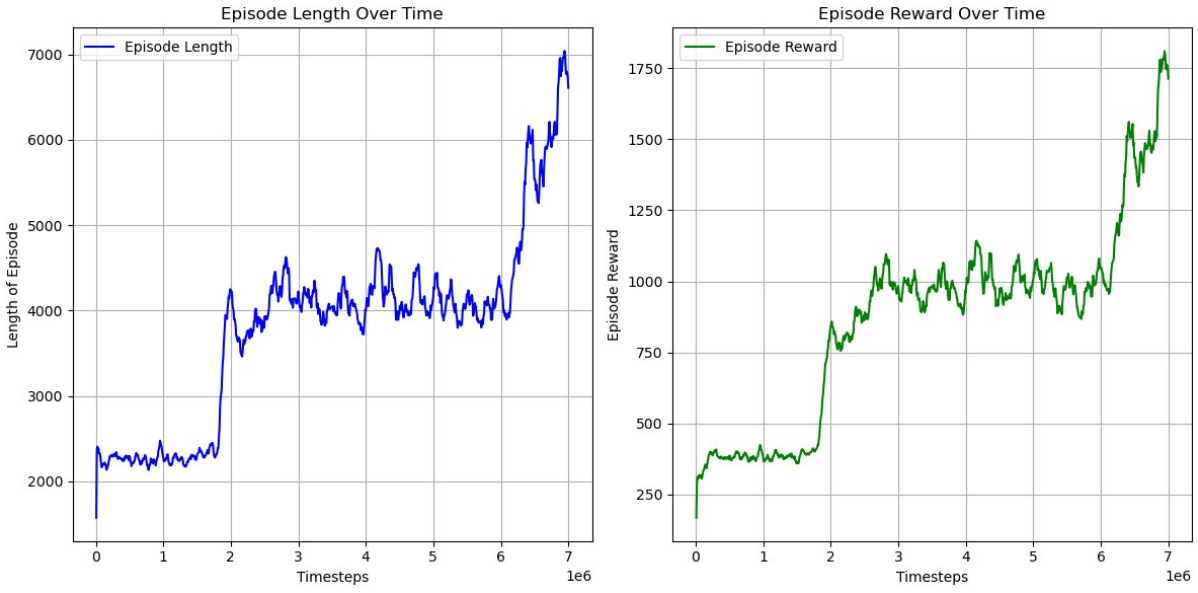
8. Sticky Actions: Introduces a small probability that the agent's previous action will be repeated, regardless of the current policy.

Also from stable baselines we use ***VecFrameStack(env, n_stack=4)***

9. Frame Stacking: Stacks a sequence of four consecutive frames to create a single observation with temporal context. The resulting input has dimensions (4, 84, 84).

We decided to train it for 5.000.000 timesteps and since it seemed that the reward and the episode length could still increase, we decided to train it for 7.000.000 timesteps, reporting all data with tensorboard. We also used the default learning rate, gamma, number of steps to run for each environment per update… That comes with the A2C function.

2.2.1.4 Results



These are our results for the final model (7.000.000 timesteps), they might seem the same plot but they're really not, normally the longer it takes you to die, the higher the reward. We decided to record a video of the agent performing from 4000 to 6000 steps and results tend to stay between 2.500-3.500 points, it reaches the 8th screen of aliens most of the times, it performs in a very human way, moving from side to side trying to kill aliens but it doesn't reach a very high score compared to PPO.

2.2.2 PPO (Proximal Policy Optimization)

In this case, the environment was set up the same way as for the Actor Critic algorithm. There is a README file in the GitHub explaining how to use the files in this part and which are the requirements needed.

2.2.2.1 How does PPO work?

It is a relatively new approach, and many refer to it as the state of the art, since it does a very good job at balancing between performance and comprehension of the environment. It is a **policy gradient method,** meaning it learns the policy function, molding how the agent generates and takes actions based on the received observation. The model returns the log of probabilities of actions in a particular state, taken based on a particular moment of the environment and the changes are reflected in the policy, based on the gradients. **Clipping the objective function is** one of the most important aspects of this algorithm, ensuring large policy updates are done and avoiding erratic outcomes. This is done by clipping the ratio between the probability of actions under the new policy and the old policy. In this model's architecture, apart from the **policy network,** often a **value network** is included, to help reduce the variance of the estimates of policy gradients and help the learning process with more information, in that case, the output is a single value (that of the expected cumulative reward).

2.2.2.2 Why use PPO?

Firstly, this algorithm is known for its simplicity, efficiency, very good results in these types of environments which are more complex, and sample efficiency. Whilst it is a fairly simple algorithm, it streamlines processes unlike other more complex models such as TRPO, and ensures the same robustness and results. The decision was mostly based on the fact that this is considered the state of the art, and it's the perfect combination of ensuring we get the best results possible and also getting an easier-to-comprehend, more stable and more efficient learning process.

2.2.2.3 Our PPO usage.

In this case, the environment set up is the following:

env = make_atari_env("Assault-v4", n_envs=4, seed=0)

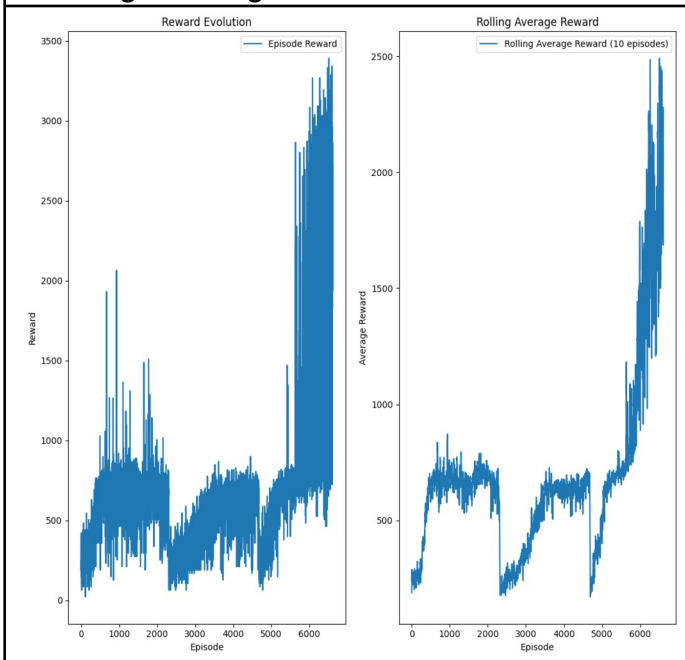
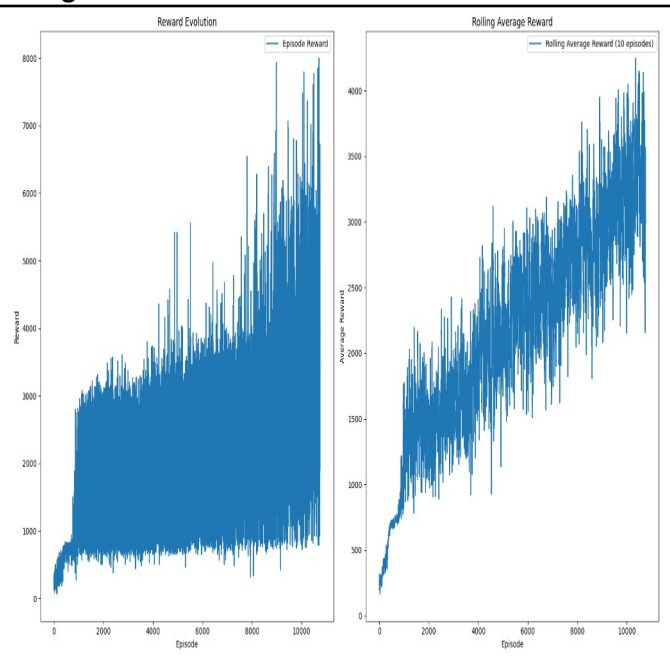
env = VecFrameStack(env, n_stack=4) #frame stacking

The only change compared to the A2C is that, in this case, the number of environments generated to train the model is 4. Moreover, to track the results along the training process, a class named **RewardTrackingCallback,** was used as our callback function when creating the model, so that at each step we can see how everything is progressing, from the reward and loss, to the hyperparameter updates that needed to be checked. The model was created using the **"CnnPolicy"** policy model, this was a fairly simple choice since we wanted to use convolutional layers and use the game's image to allow the model to learn. Further changes were made, specially focused on the way the model learned and also its stability, this will be explained in the following steps.

2.2.2.3 Solving Assault Using PPO.

In our case, we are going to be using the model provided by stable baselines, which generally ensure good results without many problems. The things that could make the learning vary a lot were: **how we set up the policy** depending on the type of space we are working with**, tuning the hyperparameters**

that can result in bigger changes, and finally **trying to tune other specifications to ensure we are doing the best we can** whilst solving specific problems we might find along the way. Our first idea was to simply run the model using various learning rates and changing n_steps when calling the function, which is the **batch size of the experience collected.** This last value, together with the batch size, results in the number of mini-batches the model uses for gradient updates. As we tried with a lower number of episodes, we realized that the training in all the cases, even though it was better using some hyperparameters above others, was very unstable in all cases. So, no matter the values used to tune and learn in different ways, the learning resulted in an ups-and-down process that was very good at some points but then went down repeatedly. We established that this was happening because the learning rate was too aggressive, so the higher values as training went on resulted in instability. Even though this was highly unstable, we realized that we had to use much smaller learning rates (in the e-5 range) to ensure more stability and better results long term. To resolve this, we decided to use a **learning rate scheduler**, to ensure that as training went on, the steps taken to update the policy were not as aggressive as those that were making the training process as unpredictable as it was. This resulted in the same results long term as we got on the spikes in the first trials, but with a lot more stability and a training process that was taking the steps at an appropriate rate.

Not using a learning rate scheduler	**Using a LEARNING RATE SCHEDULER**
	

Moreover, as we realized that this solved our main issue, what we tried to change then was mainly the learning rate at which we started and the way it progressed. We then found a value that worked very well and simply tried to do it with a larger number of episodes to see how the process progressed and if we were doing things right. We suspect that maybe using a much lower value at the beginning and without using a learning rate, we may get a similar result, nevertheless the scheduler was the first idea we had, and its execution got satisfying results.

The **number of steps**, along with other hyperparameters that we studied, resulted in better results as the number of iterations got larger. In the final model, we got to see that even after a very large number of episodes, the model still learned pretty consistently and got better at each step. We deduced that by increasing the number of training episodes and tweaking the usage of the learning

rate and its scheduler, the agent could still learn and make a considerably large positive change to its performance.

2.2.2.4 Possible Improvements.

We have seen that as we apply a higher and higher number of iterations it keeps getting better, so we could do a very large number and set a minimum value for the learning rate so that it does not get extremely low, and the model stops learning. We could also modify the reward and set a survival value at each step or simply give rewards to the agent for each bullet it dodges.

2.2.2.5 Testing and Results.

Since we wanted to be sure that our agent was learning properly and compare how well our agent behaved after training compared to the beginning, the best way to do so was to load the trained model and an untrained example to create the same number of steps and check their movements, results and ensure our process and model worked as expected. It is clear that the trained model does a lot better, the results got severely better and the way the agent moved was very reasonable and worked fairly fine, taking into account the randomness posed by the environment. We have generated a 6000-step video to showcase how the loaded model performs, and we can see that, since it is a very random environment, the range of results is still quite large, ranging from 3000 to 6000, but it still performs very well overall. We are very happy with the results.

REFERENCES:

1. https://www.researchgate.net/publication/360535268_Enhancing_reinforcement_learning_performance_in_delayed_reward_system_using_DQN_and_heuristics_February_2022
2. https://medium.com/aiguys/curriculum-learning-83b1b2221f33
3. https://openai.com/index/openai-baselines-ppo/
4. https://huggingface.co/blog/deep-rl-ppo
5. https://stable-baselines3.readthedocs.io/en/v1.0/modules/ppo.html
6. https://stable-baselines3.readthedocs.io/en/v1.0/modules/a2c.html