Reinforcement Learning Project

Shoot'Em Up

Code : https://stratus.binets.fr/s/fF88Q6mKQEA2w9i

Eyal Benaroche, Thibaut de Saivre, Matthieu Olekhnovitch

***Abstract*—This project employs Evolution Strategies (ES) to train agents in a Shoot'Em Up game, focusing on the *reverse bullet hell genre* found in games like 'Vampire Survivor'. Challenges include limited game state access and complex reward function design. Re-implementing the game in Python facilitates understanding reinforcement learning integration in diverse scenarios.**

Choosing ES over gradient-based methods accommodates inaccessible game states and simplifies parameter measurement. Beyond training agents, the project explores effective interaction with incomplete information, emphasizing adaptability in real-world environments.

Incremental project complexity aligns with Bullet Heaven game design, offering insights into Evolution Strategies' viability. In summary, this work highlights ES as an effective alternative for training agents in challenging environments, integrating innovative game design and adaptive strategies.

I. Introduction

In the realm of reinforcement learning, training agents to excel in complex and dynamic environments remains a formidable challenge. This research embarks on the exploration of Evolution Strategies (ES) as an alternative to traditional gradient-based policy methods for training agents to play a video game of the Shoot'Em Up genre, specifically a reverse bullet hell format reminiscent of the game Vampire Survivor.

The inherent difficulty of training an agent in this context arises from several challenges. First, the nature of Shoot'Em Up games often denies the agent access to the entire state of the game (the spawning position of enemies, random objects acting as buff or debuff, the power-up available in the next level...), limiting its ability to make informed decisions based on complete information. Second, the complexity of designing effective reward functions is amplified when only a set of parameters can be measured, and the agent relies on a reward signal as the sole feedback mechanism.

The project's primary objective revolves around addressing the challenge of training an agent in an infinite loop game with increasing difficulty. By re-implementing a Shoot'Em Game in python, we aim to observe the learning process of the agent as it navigates progressively challenging game scenarios. The absence of an existing gym implementation for this niche game genre has allowed us to gain more insights in the design of games in relation to the integration of reinforcement learning.

Recognizing the limitations of gradient-based policy methods for training our agent, we turn to Evolution Strategies. ES offers a promising avenue for optimizing policies without the need for gradient information, making it well-suited for scenarios where the entire state of the game is inaccessible or where measuring multiple parameters is the only feasible approach.

The project's focus extends beyond training agents; it delves into game design intricacies necessary for successful integration with reinforcement learning. As the challenge lies in developing methods to interact with both the game state and the agent's knowledge of the state in an environment where complete information is often unavailable, here as we implement the game from scratch we have a complete understanding of the game system and can easily tweak it to fit our needs. Indeed, while the real-time nature of decision-making in Shoot'Em Up games adds another layer of complexity, we allow the agent here to take time to take his decision like a turn-based game.

Moreover, the decision to re-implement a Bullet Heaven Game, strategically aligns with our goal of incrementally increasing the complexity of the project. The original design of this game genre, characterized by intense waves of enemies, dynamic behaviors, and escalating difficulty, provides a natural framework for systematically introducing challenges to the learning agent. By crafting the game design step by step (adding complexity to the game when the former design is working : bigger map, more enemies, bigger player state... see more in the background section), we have the opportunity to observe the agent's evolving capabilities in response to each augmentation in complexity. This iterative approach not only reflects the dynamic nature of real-world scenarios but also allows us to fine-tune our Evolution Strategies, reward functions, and heuristics in a controlled environment.

In summary, this project contributes to the understanding of training agents in challenging environments with limited information, emphasizing the role of Evolution Strategies as a viable alternative to gradient-based policies. By combining innovative game design approaches with adaptive training strategies, we aim to shed light on the integration of reinforcement learning in complex and evolving game scenarios.

Given more time, we would have exploited the following elements to the game being :

1) Increasing the action space : New movement like dash, dodge and reload. As the training is already quite slow, we decided not to.
2) In accordance with the reload action, an ammunition system with either random spawning on the map or a loot system with enemies
3) Change the map design (bigger map, change the geometry...) : This specific point being a bit harder to im-

plement need to rework the enemy natural path towards the player to properly overcome the potential obstacles, thus abandoned.
4) Different kind of enemies (faster enemies, dodging enemies...) : Enemy Speed is already modular as we have 2 kind of enemy, we didn't tinker too much with theses parameters, instead we worked on their size, we could have tinkered with the HP of the enemy as well to add some difficulty to the game.
5) A Hit Point (HP) system : It is implemented, but neither in use nor visible on the screen (with some kind of Heart laying in one of the corners). The current use being a way to penalize the agent during training (see details in the Results section). It can be changed in the *game_environement.py* file.
6) Items spawn on the map giving temporary buff or debuff to the player
7) Adding reward in stats for the player for either killing a specific amount of enemy as in a level-up behavior.

II. Background

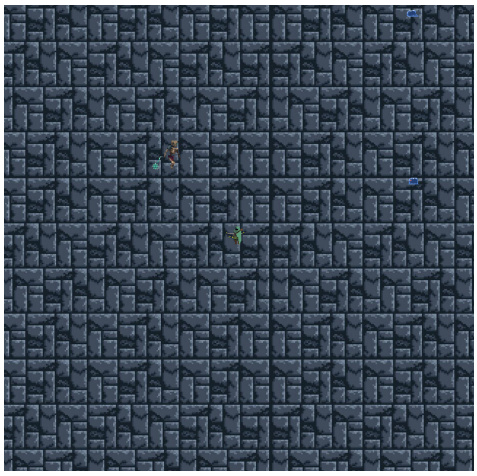

Fig. 1. Illustration of the gameplay

Shoot Em Up environment has been developed using PyGame as a baseline. The original design is kept sober, yet some animations have been added with simple assets for a better user experience.

As the game is to be modified to increase its complexity, every parameter of the game can be customized. The map composed of a blue slab tile to give it a dungeon-like aspect, the player is an archer capable of shooting simple purple dots projectiles while enemies are either skeletons (fast but big) or slimes (slow but small).

The original action space consists originally of 7 discrete actions :

1) The 4 motions : Up, Down, Left and Right (4)
2) To shoot (1)
3) The orientation of the player turning either clockwise or anticlockwise (2)

The observation state of the game s is of a variable size as the number of entities is of variable size and once again many parameters can be modified :

1) Agent Position x and y (2 floats)
2) Agent Orientation $sin(\phi)$ and $cos(\phi)$ values to provide a continuous orientation representation (2 floats)
3) Agent velocity v_x and v_y (2 floats)
4) Agent current cooldown on gun c_{player} (1 float)
5) Enemy entities positions and velocity $\{(x_i, y_i, v_{x_i}, v_{y_i})$ for $i \in [\![1, m]\!]\}$ ($4m$ floats with m changing as enemies keep spawning and dying)
6) Time since the beginning of the game (1 float)

To keep up with the variable number of enemies, we keep the observable state for the agent to $6+5\times m_{max}$ and we pad with 0 tensors to keep the agent alert to the closest enemies (more details in the methodology section).

The original naive reward function was to :

1) penalize the agent if enemies get too close
2) reward the agent when an enemy gets killed

But this has been modified as the agent should be rewarded the longer it lives as this is a survival game once and foremost. Therefore, the original model is a neural network of size $n_{observable}$ as input, h hidden layer of size 64 and output layer n_{action} of size 7 and optimized using the cma library.

III. Methodology/Approach

Our game is composed of an infinite loop, unlike most game with a clear defined state of "winning the game". Hence, the difficulty to reward the agent for his job, the goal is not to win but to last longer. It's a time based (or survival) game. Therefore, we need to keep in mind the time spent since the beginning of the game as a baseline for the reward function. But we also want to find a way to define a "*better state*" to allow us to compare the results between 2 agents without constraining the agent too much and allow exploration. Our goal is to find tailored reward functions to increase the survivability of the agent and allow midterm planning such as shooting the enemies.

A. Reward survivability

To enhance the agent's survivability in the game, we propose a modified reward function that takes into account both the agent's ability to avoid close encounters with enemies and its longevity in the game. The original naive reward function, which focused solely on penalizing the agent for close encounters and rewarding successful enemy eliminations, is insufficient for capturing the essence of a survival game.

1) Distance-based Penalty: Firstly, we maintain the penalty for proximity to enemies, but we introduce a more nuanced approach. Instead of a binary penalty, we implement a distance-based penalty that increases as the agent gets closer to

enemies. This encourages the agent to maintain a safe distance, promoting strategic positioning and movement.

Let d_{min} be the minimum distance between the agent and any enemy. The distance-based penalty P_{distance} can be defined as follows:

$$P_{\text{distance}}(d_{\text{min}}) = \frac{-\beta}{1 + e^{f(-d_{\text{min}})}}$$

$f : [0, 1] \to \mathbb{R}$ be a map function defined as follows:

$$f(d) = \begin{cases} -4 + 40d & \text{if } 0 \leq d \leq d_{threshold} \\ 4 & \text{if } d > d_{threshold} \end{cases}$$

Giving the following output :

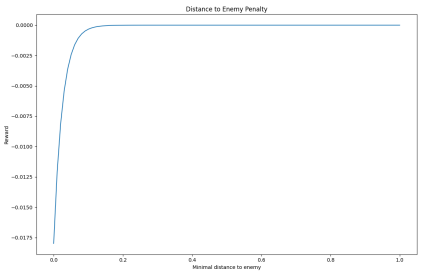

Fig. 2. Training the model while restricting shooting

Here, the sigmoid function ensures a smooth increase in penalty as the agent approaches enemies, with the threshold parameter determining the distance at which the penalty starts to escalate (here set at 20% of the map size) and β is a balancing factor that determines how much the agent should start running away. We also added a reward for the model when it stays close to the center.

2) Survival Time Reward: In addition to penalizing close encounters, we introduce a reward mechanism based on the agent's survival time. The longer the agent survives, the greater the reward it receives. This aligns with the nature of the game, emphasizing the objective of lasting as long as possible. The survival time reward R_{time} can be defined as follows:

$$R_{\text{time}}(t) = \alpha \times t$$

Where t is the time elapsed since the beginning of the game, and α is a scaling factor to adjust the relative importance of survival time in the overall reward.

3) Overall Reward Function: Combining both components, the modified reward function $R_{\text{survivability}}$ is a weighted sum of the distance-based penalty and the survival time reward:

$$R_{\text{survivability}} = R_{\text{time}} + P_{\text{distance}}$$

This reward function aims to incentivize the agent not only to evade immediate threats but also to exhibit long-term survival strategies, aligning more closely with the goals of a survival-based game. It will be subject to changes and refined further before the final submission.

B. Reward Killing

The aspect of rewarding the agent for successfully eliminating enemies poses a unique challenge, mainly because the reward for killing enemies occurs after the moment the model chooses to fire a bullet. This delay in reward makes it more difficult to directly associate actions with positive outcomes, requiring a thoughtful approach to encourage the agent to make effective offensive decisions.

1) Delayed Reward Problem: The delayed reward problem arises due to the inherent time lag between the agent's decision to shoot and the actual elimination of an enemy due to the bullet travel time. Traditional reinforcement learning models struggle with this issue, as they often attribute rewards to immediate actions, hindering the learning process for actions with delayed outcomes.

To address this challenge, we propose a reward mechanism that accounts for the delayed nature of enemy eliminations. Instead of solely relying on immediate feedback, we introduce a temporal component to the reward function, associating rewards with successful kills that happen within a certain time window after the agent's action.

2) Temporal Reward Window: Let t_{kill} represent the time at which an enemy is eliminated, and t_{action} denote the time when the agent decided to shoot. We define a temporal window Δt during which successful kills contribute positively to the agent's reward:

$$R_{\text{kill}}(t_{\text{kill}}, t_{\text{action}}) = \begin{cases} \gamma & \text{if } t_{\text{kill}} \leq t_{\text{action}} + \Delta t \\ 0 & \text{otherwise} \end{cases}$$

Here, γ is a fixed reward for a successful kill within the specified time window. This formulation allows the agent to associate rewards with its decisions to shoot, even if the positive outcome occurs slightly later.

3) Reward Oriented Shooting: While we encourage the model to have an aggressive behavior and kill the enemy, it still struggles with shooting **at** the enemy. To address the challenge of the agent struggling with orientation while shooting, we introduce an additional reward component that encourages the agent to align its shots in the right direction. This aims to enhance the agent's offensive capabilities by promoting effective targeting and orientation during engagements

The new reward component, $R_{\text{direction}}$, is based on the alignment of the agent's shots with the direction of the enemies. Let ϕ_{agent} represent the agent's current orientation and ϕ_{enemy} denote the angle of the line connecting the agent to a particular enemy. The reward is defined as follows:

$$R_{\text{direction}}(\phi_{\text{agent}}, \phi_{\text{enemy}}) = \delta \cdot \cos(\phi_{\text{agent}} - \phi_{\text{enemy}})$$

Here, δ is a scaling factor that determines the strength of the directional reward. The cosine function ensures that the reward is maximized when the agent's orientation aligns

perfectly with the direction of the enemy, encouraging the agent to face its targets while shooting.

4) Incorporating Directional Shooting and killing Reward into Overall Reward: The total reward function now includes the survivability reward, killing reward, and the newly introduced directional shooting reward:

$$R_{\text{total}} = R_{\text{survivability}} + R_{\text{kill}} + R_{\text{direction}}$$

Balancing the weights of these components is crucial to achieve a well-rounded and effective learning process. Fine-tuning the scaling factors and parameters will be necessary to ensure that the agent not only survives but also exhibits strategic shooting behavior.

IV. Results and Discussion

We will now discuss the influence of the different reward functions and parameters of the models for the training process and the results it gives.

A. Aiming policy

Here we train the model only to specifically aim at a target and restrain its behavior, removing the possibility of movement and using jointly the aiming and killing reward functions, making it act as a turret.

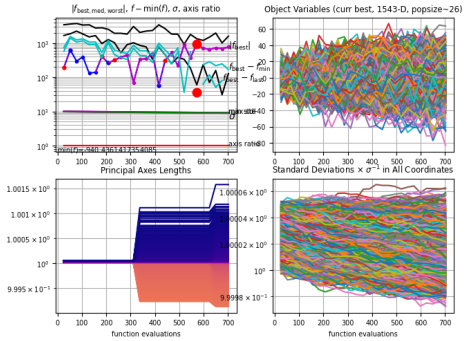

Fig. 3. Training the model while restricting its movements

While the model doesn't survive for long as the agent gets overwhelmed when too many enemies are close and struggle to get it's priority straight (shooting the closest enemy as it's the most dangerous and then by order of distance), we still have a model capable of surviving for a couple of seconds, and it doesn't converge yet to a shooting machine with perfect precision as we expected. (see the resulting gifs output in the code base)

B. Escaping Policy

While it seemed like a good idea originally, the time based policy reward is not a keeper as the reward value goes out of hand too fast. We lack time to explore a way to diminish its influence while still helping our model to keep its surviving behavior, but we decided instead to give the agent infinite HP during the training (it doesn't stop to move after being hit) but instead heavily penalize every collision. This time, to assess the influence of the escaping policy, the model cannot shoot at the enemies.

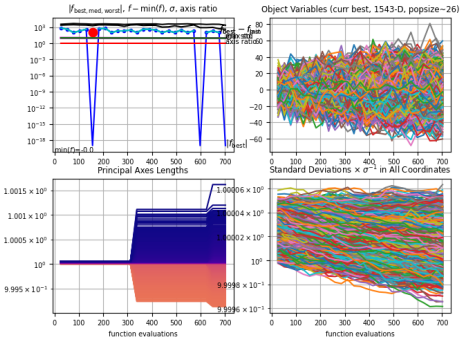

Fig. 4. Training the model while restricting shooting

This time the training doesn't give good results and converges towards an agent running away blindly instead of avoid enemy. A failure giving insight towards the poor capacity of the model to avoid the enemies, something a more experienced player can do quite easily (see gifs output as well).

C. Aggressive Policy

Now, we remove the distance penalty reward and watch how the model interacts with the environment.

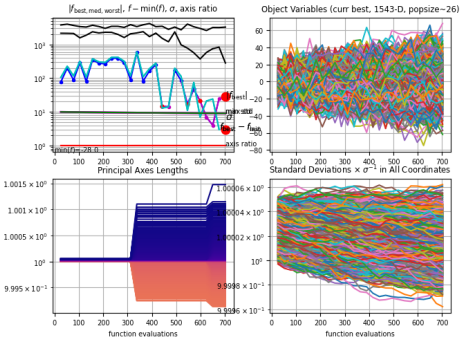

Fig. 5. Training the model with all rewards and actions

This time again, the model struggle to survive as it keeps running into the enemies blindly, while it does manage to kill some of them before dying as it keeps shooting in front of him, therefore attaining results better than the complete escaping policy, this is definitely not the intended behavior as this kind of games should reward more the ability to survive for longer period of time instead of killing more enemies in a short time frame.

D. All rewards at once

Finally, with all rewards active, we run another training session giving these results :

While in the first training sessions, our agent created a 'hide and shoot' strategy of the player, running to a corner and shooting desperately in front of him without any aim. We

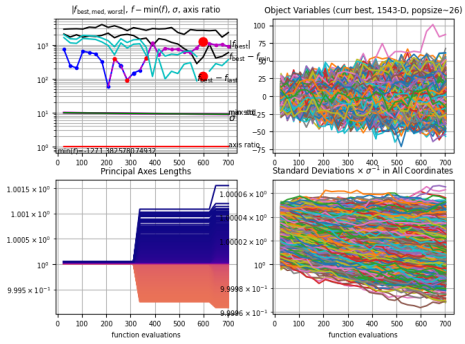

Fig. 6. Training the model with all rewards and actions

now have an agent playing the game, but it is still struggling at shooting properly and knowing where to go when cornered by many enemies. The distance based policy could be improved quite a lot to better teach the model how to escape his enemies. The reward functions weights could as well be tinkered with to enhance the agent's performances.

It would be quite interesting to do other training sessions while giving the model more awareness of its surroundings and checking the influence of the size of the hidden layer for the model neural network.

E. Specialized Model for Aiming

Recognizing the challenges faced by the current agent in accurately orienting itself for shooting, we consider a targeted approach by training a dedicated model for aiming purposes. Rather than employing a comprehensive model for all action states, this specialized model focuses solely on refining the agent's aiming capabilities.

1) Optimal Angle Computation: In the initial phase, we adopt a straightforward approach by computing the optimal shooting angle using only the positions of the agent and the closest entity. This trigonometric approach gives the angle ϕ_{opt} that aligns the agent with the target entity.

2) Neural Network Relaxation: To further enhance adaptability and handle more dynamic scenarios, we explore the possibility of relaxing the initial rigid approach by introducing a neural network. This neural network is designed to predict the shooting angle, taking into account not only the positions (x, y) but also the velocities of both the agent and the target entity (the observable environment state). The introduction of velocity information adds a dynamic component to the aiming process, allowing the agent to adapt to changing scenarios.

The joint training of this aiming neural network with the original model promotes the integration of aiming capabilities into the agent's overall decision-making process. The neural network learns to predict shooting angles based on a broader set of parameters, incorporating velocity information for both the agent and the entity.

3) Training Objective: As we lacked of time, it would have been great to train it to compute the optimal angle ϕ_{opt} given the entire observable state s and then incorporate it to the original model following Fig7 and a joint training approach.

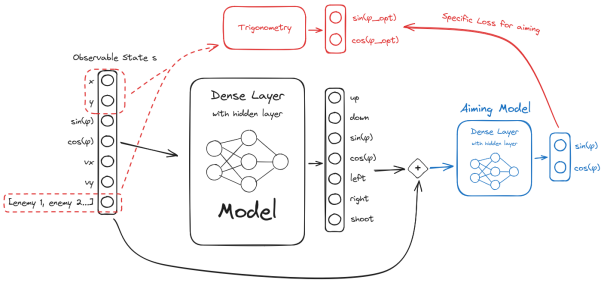

Fig. 7. Initial model design

F. An alternative to tackle the temporal delay issue : LSTMs

As we encountered some difficulties with the model, dealing with time projections and delay managing, we chose to add a temporal component to the reward. However, we could have approached this part of the issue differently using another architecture for our neural network : Long short-term memory (LSTM). Such an implementation can help us learn the temporal dependencies between player actions and enemy responses. To do so, we tried several combinations of hyperparameters, letting the player remember the last 10-20-30 frames, for example. It seemed that the model was more adaptable, but it required significant computational resources and training time as we increased the frames in the neural network 'memory'. Concerning the results, this LSTM implementation demonstrated greater adaptability to varying game scenarios, but fewer survival time results compared to our initial approach. However, further optimization of hyperparameters and training strategies may be necessary to fully leverage the potential of LSTM in addressing the temporal aspects of these gameplay dynamics. Due to the limitations in computational resources, we found that our initial approach remained highly effective and within the scope of our project objectives. While it first showed promise in addressing temporal dependencies, the computational demands associated with training and optimization of the LSTM presented significant additional challenges. Therefore, we focused our efforts on maximizing the efficiency and effectiveness of our initial approach, which proved to be a pragmatic decision given the constraints.

V. Conclusions

In conclusion, this project has proven the effectiveness of Evolution Strategies (ES) in training agents to navigate and discover challenging environments, particularly within the context of Shoot'Em Up games. By re-implementing the game in Python, we acquired valuable insights into the integration of reinforcement learning techniques and the complexity of game design itself. When developing and enhancing the reward function, we discovered the importance of its high sensitivity to the nuances of gameplay for effective training. However, such a sensibility comes with its set of challenges, and among them, computation power,

as exploring numerous solutions requires significant resources.

Moreover, we found that while heuristic can indeed influence agent behavior towards a specific goal, they can also restrict variability and limit the adaptability in diverse scenarios. Our exploration also shed the light on the fact that many off-the-shelf solutions were not giving good results, as they were too generic and deeply unadapted to the specific environment. To go even further, the next step for research would be to design a fully deterministic model that could outperform the model presented in this paper. Such model could show more precise actions such as shooting at perfect angles or move towards low density regions.

Lastly, the development part of this project highlights the complexity and importance of game design. In essence, our findings underscore the intricate interplay between reinforcement learning (RL), heuristic strategies, and game design fundamentals. Moving forward, addressing these challenges in a good way may be crucial for advancing the frontier of AI and autonomous agents in controlled environments such as video games.

References

[1] Sutton and Barto. Reinforcement Learning, *MIT Press*, 2020.
[2] Read. Lecture IV - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.
[3] Wikipedia, *Shoot'em Up* https://en.wikipedia.org/wiki/Shoot_%27em_up, accessed 10 March 2024.
[4] Poncle, *Vampire Survivor*, https://store.steampowered.com/app/1794680/Vampire_Survivors/, accessed 11 March 2024.
[5] Jerin Paul Selvan, *Playing a 2D Game indefinitely using NEAT and Reinforcement Learning*, arXiv:2207.14140, Jul 2022

Appendix

The Code is available at the link below the title. Please unzip it and follow the *README.md* file for more instructions