UNIVERSITÀ DI PISA

Informatic Engineering Department

Artificial Intelligence and Data Engineering

Non-Archimedean Reinforcement Learning: A deep approach to a more complex benchmark (Atari games)

Students:

Lorenzo Massagli

Antonio Patimo

Academic Year 2022/2023

1 Introduction

Reinforcement learning (RL) is a subfield of machine learning that involves an agent working towards goals within a given environment by executing appropriate actions. Each action undertaken by the agent alters the state of the environment and generates a reward indicative of the effectiveness of the executed action, which is received by the agent. By understanding the reward obtained from doing a particular action in specific environmental states, the agent tries to develop an optimal action plan, referred to as a policy. This policy is considered optimal when it ensures maximization of the expected discounted reward over time.

Most studies within this area typically assume that the agent is focused on a singular objective, with the reward being represented as a simple numerical value. However, there is a lot of problems spanning across various sectors that have multiple objectives. In our study, we focus on a category of multi-objective problems where a hierarchy is established among the objectives. This hierarchy enforces an order of priority for optimizing each objective. A problem framed in this manner is called as Lexicographic Multi-Objective Problem (LMOP).

One potential approach to executing Lexicographic Multi-Objective Reinforcement Learning (LMORL) tasks is to manage multiple rewards through a process called scalarization. It consists in weighting each objective function with respect to their priority. The issues with this approach are that there is not guarantee of problem equivalence with the original one, because the weights are chosen arbitrary and are error-prone (trial-and-error approach), and that there is not guarantee of converge to the global optimum of the problem.

The non-Archimedean scalarization leverages a scalarization method that involves infinite and infinitesimal weights. In practice, reformulates the multi- objective problem as a single (non-Archimedean) objective one still guaranteeing the satisfaction of the lexicographic ordering; this means that, in principle, we can solve the reformulated problem using common RL techniques, once tailored to cope with

non-Archimedean values. This approach guarantees the convergence to the optimal solution (if any) and that the problem is equivalent with the original one. The problem formulation can be seen in the equation 1.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f_1(x)\alpha^0 + \ldots + f_n(x)\alpha^{1-n} \\ \text{subject to} \quad & x \in D \end{aligned} \tag{1}$$

In this work we will explore the application of the non-Archimedean scalarization to LMORL (Lexicografical Multi-Objective Reinforcement Learning) problems, solving them leveraging a non-Archimedean Deep Q-Learning (NA-DQL) approach, i.e., a deep neural network enhanced to manipulate non-Archiemdean values. The problems that will be taken into account are the Atari games, which are a complex benchmark in the Reinforcement Learning field.

The goal of this work is to analyze the different performance between a standard Atari game RL model and a non-standard hybrid model approach (non-Archimedean).

2 Related Works

This work refers to the thesis of Dr. Silvestri, who applied the non-Archimedean Deep Q-Learning approach to two RL benchmarks: Lunar Lander v2 and Mountain Car.

Lunar Lander v2 is a benchmark or simulation environment commonly used in reinforcement learning (RL) research. In this scenario, the task is to control a spacecraft as it attempts to land safely on the surface of the moon. The spacecraft is subject to the forces of gravity, thrust from its engines, and various obstacles such as craters and uneven terrain. In the thesis it has been described that the Non-Archimedean Deep Q-Learning (NA-DQL) agents achieved the average cumulative reward to solve Lunar Lander v2, learning to fly and land safely. Challenges included initial crashes and unfavorable conditions, but over 90% of agents with optimized hyperparameters succeeded. GC-NA-DQL (Gradient Clipping NA-DQL) agents performed best, reaching high success rates (87.5%) and stable performance, outperforming standard agents and GC-Hybrid agents. Optimization focused on descent quality and fuel consumption, demonstrating effective learning strategies.

Mountain Car is a popular benchmark and simulation environment used in reinforcement learning (RL) research. In this scenario, there is a car situated in a valley between two mountains. The objective of the car is to reach the top of the right mountain by applying appropriate acceleration and deceleration actions. In the thesis it has been described that Standard DQL agents struggled to solve the Multi-Objective Mountain Car (MOMC) environment, often getting stuck in suboptimal solutions that prioritize minimizing fuel consumption over reaching the goal. However, when using non-Archimedean scalarization, the behavior of the agents improved significantly. They learned to prioritize reaching the goal and were willing to spend fuel in the early stages of learning. Lower learning rates ($lr = 10^{-}5$) proved to be effective, leading to outstanding performances. Both naive-NA-DQL and GC-NA-DQL agents achieved similar results, successfully solving the environment

in most cases. Hybrid agents performed slightly better but required more training episodes on average. The successful episodes showed consistent goal-reaching behavior with optimized fuel consumption.

3 Alpha Theory

Non-Archimedean analysis is a mathematical branch that deals with fields lacking of the Archimedean property. We will focus on the Euclidean numbers and their axiomatization called Alpha Theory. This axiomatization is composed by just three axioms.

Axiom 1 (Existence)**.** *There exists an ordered field of $E \supset R$ whose numbers are called Euclidean numbers.*

Before introducing the second Axiom, we need the following definition that introduces a partitioning of $\mathbb{E}$ in three categories: infinite, finite and infinitesimal numbers.

Definition 1 (4.1)**.** *Let $\xi \in E$. Then:*

- *ξ is infinite $\Leftrightarrow \forall n \in \mathbb{N}, |\xi| > n$,*
- *ξ is finite $\Leftrightarrow \exists n \in \mathbb{N} : n^{-1} < |\xi| < n$,*
- *ξ is infinitesimal $\Leftrightarrow \forall n \in \mathbb{N}, |\xi| < n^{-1}$.*

Axiom 2 that introduces the infinite number α and states that it can be manipulated as any other real number using field properties such as commutativity, associativity, etc. In this Axiom $|\cdot|$ denotes the cardinality of a set.

Axiom 2 (Numerosity of α)**.** *There exists a function $num : U \to E$ which satisfies the following properties:*

- *if A is finite, $num(A) = |A|$,*
- *$num(A) < num(B)$ if $A \subset B$,*
- *$num(A \cup B) = num(A) + num(B) - num(A \cap B)$,*
- *$num(A \times B) = num(A) \cdot num(B)$,*

- $\alpha = num(\mathbb{N})$.

Since α is the number of elements within N, the fact that the latter is an infinite set implies that α is infinite. Moreover, it is typical to define also the value η reciprocal of α, i.e., $\eta := \alpha^{-1}$, which means that η is an infinitesimal since it is the reciprocal of an infinite value. The second property defines total ordering for the field E. The third and fourth properties of Axiom 2 assert that E contains all the algebraic manipulations of α.

Now we are ready to introduce the notion of α-limit in Axiom 3.

Axiom 3 (α-limit)**.** *Every sequence $\varphi : \mathbb{N} \to \mathbb{R}$ has a unique α-limit, denoted by $\lim_{n\uparrow\alpha} \varphi(n)$, which satisfies the following properties:*

- *if $\xi \in E$, then there exists a sequence $\varphi : \mathbb{N} \to \mathbb{R}$ such that $\xi = \lim_{n\uparrow\alpha} \varphi(n)$,*
- *if $\varphi(n) = n$, then $\lim_{n\uparrow\alpha} \varphi(n) = \alpha$*
- *if $\exists n_0 \in \mathbb{N}$ such that $\forall n \geq n_0, \varphi(n) \geq \psi(n)$, then $\lim_{n\uparrow\alpha} \varphi(n) \geq \lim_{n\uparrow\alpha} \psi(n)$,*
- *for any sequences φ and ψ,*

$$\lim_{n\uparrow\alpha} \varphi(n) + \lim_{n\uparrow\alpha} \psi(n) = \lim_{n\uparrow\alpha} (\varphi(n) + \psi(n)),$$
$$\lim_{n\uparrow\alpha} \varphi(n) \cdot \lim_{n\uparrow\alpha} \psi(n) = \lim_{n\uparrow\alpha} (\varphi(n) \cdot \psi(n)).$$

The symbol "$n \uparrow \alpha$" is used to distinguish the α-limit differs from the usual limit of a sequence "$n \to \alpha$". The first and important consequence of Axiom 3 is that every real function $f : \mathbb{R} \to \mathbb{R}$ can be extended to an Euclidean function $f^* : E \to E$ by setting:

$$f^* \left(\lim_{n\uparrow\alpha} \varphi(n) \right) = \lim_{n\uparrow\alpha} f(\varphi(n)).$$

Therefore, the α-limit can be seen as a weak form of what is called the Transfer Principle, through which we can transfer first-order properties of functions to the

Euclidean field. The Transfer Principle states that for any object A defined over $\mathbb{R}$ exists a unique object ${}^*A$ over E satisfying all the first-order properties of A.

3.1 Algorithmic Numbers

Algorithmic numbers (ANs) constitute a subset of E, which can be better standardized in order to being easily used for computations on a machine. The definition of AN leverages the concept of monosemium.

Definition 2 (Monosemium)**.** *A number $\xi \in E$ is called a monosemium iff:*

$$\xi = r\alpha^p, \text{ with } r \in \mathbb{R},\ p \in \mathbb{Q}.$$

Definition 3 (Algorithmic number)**.** *A number $\xi \in E$ is called algorithmic if it can be represented as a finite sum of monosemia:*

$$\xi = \sum_{k=1}^{l} r_k \alpha^{s_k}, \text{ with } r_k \in \mathbb{R},\ s_k \in \mathbb{Q},\ s_k > s_{k+1}.$$

Definition 4 (Normal Form)**.** *An algorithmic number can always represented in a form, called normal form, such that:*

$$\xi = \alpha^p P\left(\eta^{m_1}\right),$$

where $\eta := \alpha^{-1}$, $p \in \mathbb{Q}$, $m \in \mathbb{N}$ and $P(x)$ is a polynomial with real coefficients such that $P(0) \neq 0$.

The problems with the Algorithmic numbers as defined is that this subset is not closed with respect to inversion, meaning that when computing the inverse of an AN the result is not always an AN and this representation has a variable length coding (we need a fixed length representation). To solve both these issues, we need to define the truncation operation for algorithmic numbers, in particular for the polynomial $\dot{P}()$ of the normal form for ANs.

Definition 5 (Truncation of AN)**.** *Let* $P(x) = p_0 x^{z_0} + \ldots + p_m x^{z_m}$ *with* $z_{i-1} < z_i$ *and* $i = 1, \ldots, m$*, then we define the truncation as follows:*

$$tr_n[P(x)] := \begin{cases} P(x) & \text{if } n \geq m, \\ p_0 x^{z_0} + \ldots + p_n x^{z_n} & \text{if } n < m. \end{cases}$$

This operation gives us a finite length approach called Bounded Algorithmic Numbers.

3.1.1 Bounded Algorithmic Numbers

The Bounded Algorithmic Numbers (BANs) are a particular subset of ANs suitable for computer computations. A BAN is an algorithmic number with $s_k \in \mathbb{N}$ that can be represented by the same normal form $\alpha^p P(\eta)$, with its manipulations being equipped with the truncation operation.

3.1.2 Ban Library

In order to perform computations using non-Archimedean quantities, we exploited a Library for BANs implemented in Julia. This library declares an abstract type representing Algorithmic Numbers as a subtype of Number, that is used to build the concrete type Bounded Algorithmic Number. The Bounded Algorithmic Number is defined according to its normal form, i.e., as a vector of reals representing the monosemia coefficients and an integer number representing the magnitude. Functions for various operations on BANs such as arithmetic operations, random number generation, ordering functions and linear algebra operations have been implemented in the library. Most of these functions are extensions of already existing procedures, thus allowing us to manipulate BANs as we would do with floating point numbers while coding.

4 Deep Q-Learning

Deep Q-Learning (DQL) is a variation of Q-learning (a famous RL algorithm) that admits continuous state spaces. It approximates the Q-function using a neural network, the so called Deep Q-Network (DQN).

In order to make the learning samples not correlated, we use the concept of Experience Replay Buffer (ERB). An Experience Replay Buffer (ERB) is a collection of tuples, each designated as (s, a, r, s', a'), representing an experience made by the learning agent. To prevent learning from consecutive series of transitions, these tuples are sampled, thus generating uncorrelated data. Two prevalent sampling strategies exist: random sampling and prioritized sampling. In random sampling, samples are selected arbitrarily from the ERB. On the other hand, prioritized sampling allows weighting of samples in the ERB using diverse criteria, providing a guided approach to the sampling process.

The Deep Q-Learning (DQL) algorithm utilizes an old, fixed version θ^- of the trained Deep Q-Network (DQN) θ, referred to as the Target Network, to compute target Q-Values in the loss function. This helps to avoid learning instability, which can occur if the same DQN used for predictions is also used to generate the ground truth. The loss function is defined as:

$$L(\theta) = E_{(s,a,r,s')}\left[(r + \gamma \max_{a' \in A} Q_{\theta^-}(s', a') - Q_\theta(s, a))^2\right]$$

The Target Network θ^- is periodically updated using the parameters of network θ according to the following equation:

$$\theta^- = \theta \times \tau + \theta^- \times (1 - \tau), \quad 0 < \tau \leq 1$$

This update is called a full update when $\tau = 1$, and is typically performed at the end of each episode. By decreasing the value of τ, we get what is known as a soft update, which is usually executed more frequently. For instance, a soft update

could be performed after each batch training with $\tau = 0.001$.

The classical DQN loss function is prone to maximization bias, since we use network θ^- to compute both the target action a' and the target value $Q_{\theta^-}(s', a')$. To address this issue, a variant known as Double DQN uses a different loss function:

$$L(\theta) = E_{(s,a,r,s')}\left[\left(r + \gamma Q_{\theta^-}(s', \arg\max_{a' \in A} Q_\theta(s', a')) - Q_\theta(s, a)\right)^2\right]$$

Here, the target action is selected as the greedy action through network θ, and is then used to compute the target Q-value following the policy used by network θ^-.

5 Experimental Setup

In this chapter we will present the environment (Atari Pong Game) and the agent that has been used for the experiments. Later, we will give an insight of the implementation that has been done in Python and Julia, explaining the differences in the two approaches.

5.1 Atari Pong Environment

Pong is a classic Atari game where two players control paddles on opposite sides of a virtual table, simulating a table tennis match. The objective is to hit the ball back and forth, with the goal of outscoring the opponent by making the ball pass their paddle. The game is played in a 2D environment with a top-down view. The ball moves across the screen, bouncing off the walls and the paddles. Each player can move their paddle up or down to intercept the ball. When a player fails to return the ball, the opponent scores a point. The game continues until a predefined score limit is reached, typically 21 points.

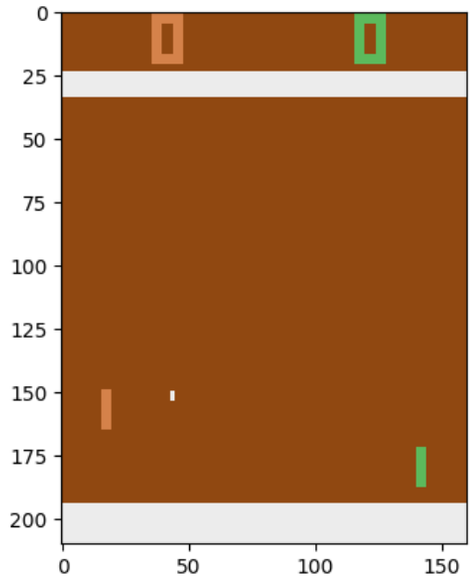


Figure 1: Pong game.

Gym Library OpenAI's Gym is a Python library that provides a simple and standardized interface to a large collection of reinforcement learning (RL) environments. It was designed to standardize the way RL algorithms interact with environments, thus allowing researchers and developers to focus more on designing and testing RL algorithms rather than worrying about how to interface with each unique environment.

In the Gym's environment the possible actions that can be taken from the agent are, to do nothing (NOOP and FIRE), to move right (RIGHT and RIGHTFIRE) and to move left (LEFT and LEFTFIRE). However, we decided to limit the number of actions of the agent to 4, because the RIGHTFIRE and LEFTFIRE where useless because they have been respectively mapped to the RIGHT and LEFT button and this redundancy could introduce some noise. The FIRE button is instead have been used to lunch the game after an episode finishes.

Num	Action
0	NOOP
1	FIRE
2	RIGHT
3	LEFT
4	RIGHTFIRE
5	LEFTFIRE

Num	Action
0	NOOP
1	FIRE
2	RIGHT
3	LEFT

Table 1: On the left the full action space, on the right the reduced one.

The state space is by default an RGB image which provides visual information about the current state of the game, including the positions of the paddles and the ball. The dimensions of the screen can vary, but it typically has a size of 210 pixels in height and 160 pixels in width. This original state space has been modified using some transformations to return a gray-scale image of 84x84 pixel to simplify and reduce the complexity of training the model.

The default reward of the game is $+1$ if the ball pass the opponent's paddle and -1 if the ball passes your paddle. We modified this reward in the multi-objective case to be a vector composed by two components: the first one related to

the score and the second one related to the number of movements that the paddle does (all the action that are not 0 or 1 gives a reward of -1). This multi-objective representation is the goal of this study because we want to solve a Lexicographical Multi-objective Reinforcement Learning problem where the agent has to maximize the first objective, the score, but wants also to minimize as much as possible the number of movements of the paddle, that is the second objective.

5.2 Atari Pong Agent

The Pong-agent is a DQN agent that uses a deep neural network to approximate the Q-function. The network is a convolutional neural network (CNN) composed by three convolutional layers, followed by two fully connected layers. The input to this network is the current state (the processed game screen), and the output is a value for each possible action. This is a standard architecture, figure 2, used in litterature, and the parameters value for each layer can be seen in table 2. The parameters used for the agent are described in table 3.

Layers	**Filter**	**Input**	**Output**	**Stride**	**Act. Fun.**
Conv_1	(8,8)	4 (stack_size)	32	4	ReLU
Conv_2	(4,4)	32	64	2	ReLU
Conv_3	(3,3)	64	64	1	ReLU
Dense	-	3136	512	-	ReLU
Dense	-	512	numactions	-	-

Table 2: DQN Architecture.

5.3 Python implementation

The environment and the agent have been first implemented in Python and later in Julia. The Python implementation has been done using mainly the PyTorch and Gym libraries. The PyTorch library has been used to implement and train the DQN agent network used by the agent to examine the state and decide which action to take. The Gym library has been used to interact with the environment and also to

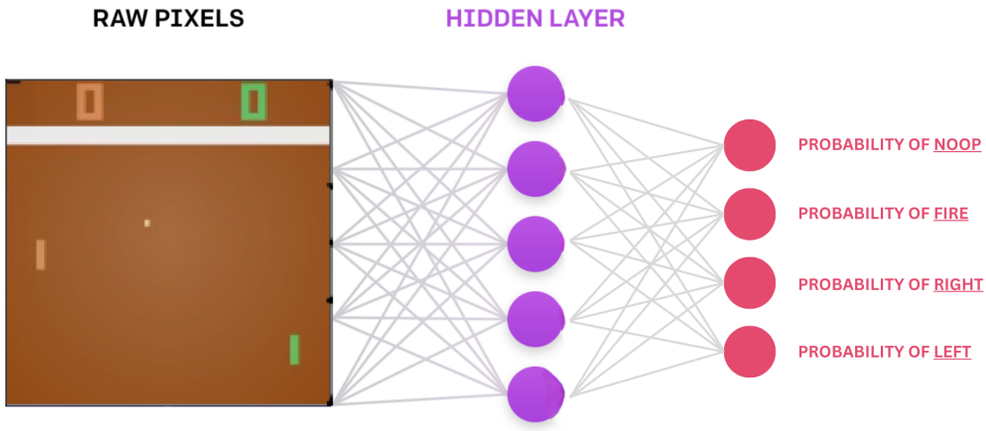


Figure 2: DQN Architecture representation.

Parameter	**Description**
γ	Discount Factor
ϵ	Epsilon for ϵ-greedy approach
ϵ_{decay}	Batch size used for training
ϵ_{min}	Minimum value for the epsilon
τ	Soft update parameter
Learning rate	Learning rate
Max Memory	Maximum size of the replay memory
Train Start	Number of experiences before starting the training
Batch Size	Size of the batch
Optimizer	Optimizer used for training
Loss Function	Mean Squared Error

Table 3: Agent parameters.

modify its state and behaviour applying some Gym's wrapper. This are the custom wrappers implemented on top of the base environment Gym's class to obtained the preprocessed state:

- FireResetEnv: Wrapper used to start the game by pressing the fire button.
- MaxAndSkipEnv: Wrapper used to skip frames and to take the maximum value of the last two frames. The skip is used to speed up significantly the training by applying max to N observations (four by default) and returns this as an observation for the step, the maximum value of the last two frames is used to reduce the flickering effect that affect some old atari games. For the

human eye, such flickering is not visible, but it can confuse a Neural Network.

- ProcessFrame84: Wrapper used to rescale the state represented by the image to 84x84 pixels and applying a gray-scale filter.

- BufferWrapper: Wrapper used to stack the last 4 frames. The state becomes a 84x84x4. We need to use the stacked frame because the DQN needs to understand the direction and speed of the ball.

- ImageToPyTorch: Wrapper used to change the shape of the frames from 84x84x4 to 4x84x84. It is a permutation wrapper to make the images suitable as input of the PyTorch network standard.

- ScaledFloatFrame: Wrapper used to normalize the frames (pixels between 0 and 1). It is used to have a better representation of the pixel, more suitable for the DNN field.

- RewardWrapper: Wrapper used to change the reward from the default one (scalar) to a vectorial one (multi-objective). This wrapper changes the reward from the scalar "score" to the vector [score, movement]. The code can be seen in the listing 1.

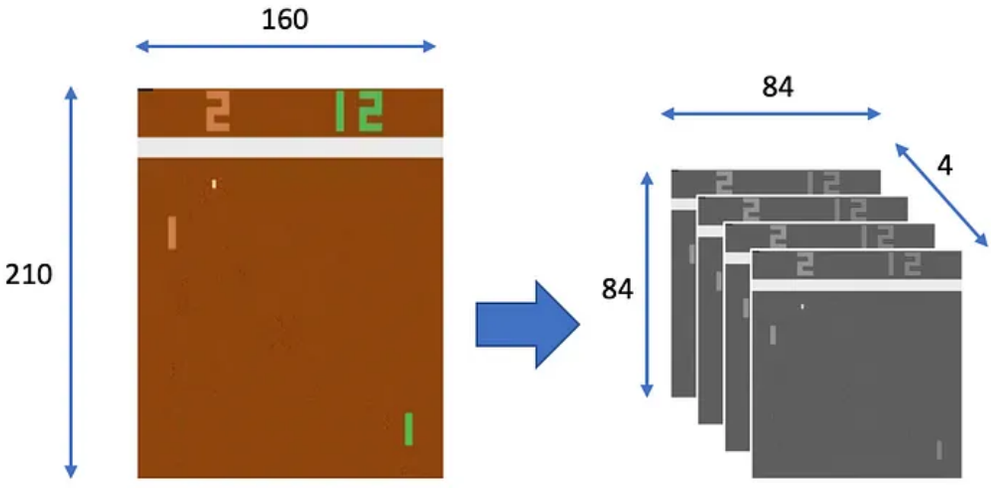


Figure 3: ProcessFrame84 and BufferWrapper results.

In Python we implemented the standard DQN agent (for the standard pong game, without env reward changed) and also a scalarized DQN agent which applies a scalarization of the reward vector. The scalarization has been obtained changing the reward of the environment using the RewardWrapper, talked about before, represented in listing 1. This wrapper let us modify the default step function of the environment to return a vectorial reward instead of a scalar one. The returned reward is processed by the agent that performs the scalarization using two weights, α_1 and α_2, and makes the weighted sum. The aim of the scalarization is to give a heavier weight to the score and give a lower weight to the movement penalization, to make the agent focus on making points but also to take in consideration the number of movements done.

```
class RewardWrapper(gym.Wrapper):
    def __init__(self, env):
        super().__init__(env)

    def step(self, action): # Change the reward function from scalar to vector
        obs, rew, done, info = self.env.step(action)

        if action != 0 and action != 1:
            rew = [rew, -1]
        else:
            rew = [rew, 0]

        return obs, rew, done, info
```

Listing 1: Reward Wrapper implementation

The agent in Python has the architecture described in table 4.

Layers	**Filter**	**Input**	**Output**	**Stride**	**Act. Fun.**
Conv_1	(8,8)	4 (stack_size)	32	4	ReLU
Conv_2	(4,4)	32	64	2	ReLU
Conv_3	(3,3)	64	64	1	ReLU
Dense (std. weights)	-	3136	512	-	ReLU
Dense (std. weights)	-	512	numactions	-	-

Table 4: DQN Architecture in Python.

5.4 Julia implementation

In the Julia implementation, differently from Python, we couldn't use the Gym library (because in Julia it doesn't support the Pong game) and the PyTorch library (because it doesn't exist in Julia). To overcome the lack of the Pong environment in the Julia Gym library, we decided to use the same environment used in the Python implementation. To do this from the Julia code, we used the PyCall package that allowed us to call the custom make_env function in Python, that returns a Python object in Julia and call the respective methods on it, for example the **env.step()** and **env.reset()**. For what concern the DQN, we used a Julia equivalent package to PyTorch, called Flux.

Flux.jl Flux.jl is a machine learning library for the Julia programming language. It's designed to be simple, flexible, and powerful, with a focus on allowing you to define complex models directly in Julia without needing to rely on separate configuration files or domain-specific languages.

With this two libraries we implemented the whole logic behind the Pong Agent and the interaction with the environment in Julia. In this case, we implemented only the non-standard version of the RL problem, using the BAN_s3_isbits library wrote in Julia to cope with the BAN numbers. The DQN implementation in Julia is hybrid, in the sense that it uses standard weights for the convolutional part of the model and non-standard (BAN) weights for the dense layers used to get the output of the network, as shown in table 5.

Layers	**Filter**	**Input**	**Output**	**Stride**	**Act. Fun.**
Conv_1	(8,8)	4 (stack_size)	32	4	ReLU
Conv_2	(4,4)	32	64	2	ReLU
Conv_3	(3,3)	64	64	1	ReLU
Dense (BAN weights)	-	3136	512	-	ReLU
Dense (BAN weights)	-	512	numactions	-	-

Table 5: DQN Architecture in Julia.

6 Experiments and Results

In this section will be described the experiment that has been done, explaining the results obtained with the Python and Julia implementations.

6.1 Python Results

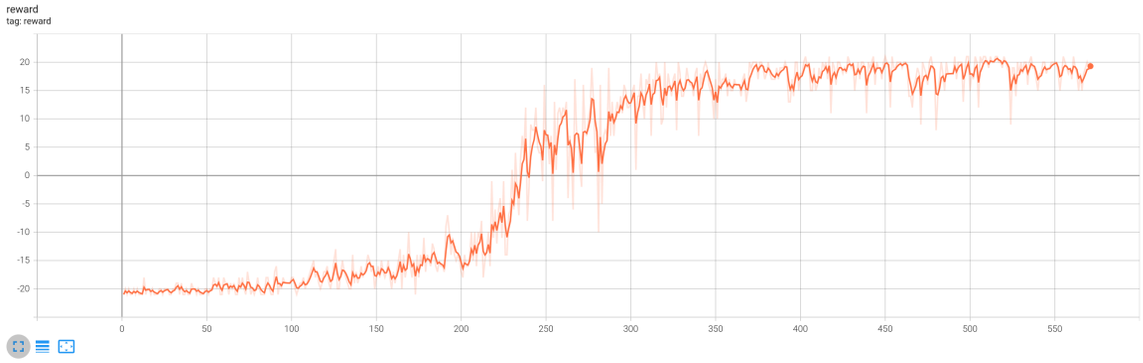
We started the experiment phase from the Python implementation, trying to training an Agent that was successful for the standard and scalarized Atari Pong game environment. We considered an average cumulative reward threshold of 18.5 on the last 100 episodes, to consider the environment solved. The parameters that has been used for the Python's DQN agents are described in table 6.

Parameter	**Value**	**Description**
γ	0.99	Discount Factor
ϵ	1	Epsilon for ϵ-greedy approach
ϵ_{decay}	0.995	Batch size used for training
ϵ_{min}	0.02	Minimum value for the epsilon
τ	0.001	Soft update parameter
Learning rate	0.0001	Learning rate
Max Memory	25000	Maximum size of the replay memory
Train Start	5000	Number of experiences before starting the training
Batch Size	32	Size of the batch
Optimizer	Adam	Optimizer used for training
Loss Function	MSE	Mean Squared Error

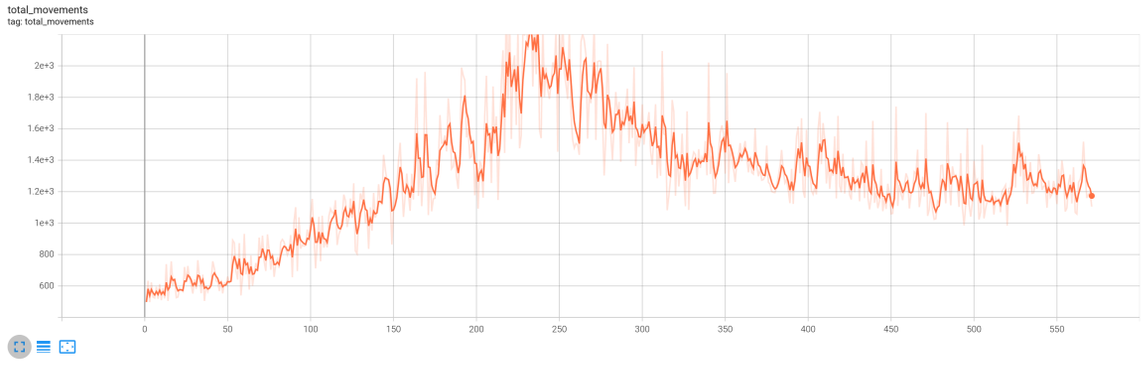
Table 6: Agent parameters.

Taking in the consideration the agent with the scalar reward (the standard one) it reached the cumulative reward threshold value in 571 episodes with an average number of movements of 1227.64. Looking at the agent at work, we saw that the agent learned a specific move that was effective to make a lot of points without getting once back. The training results can be seen in figure 4.

As second experiment we modified the standard environment, getting a vector of rewards instead of the single one representing the score. As mentioned in the section 5, the reward has been modified from the scalar value representing the score to a



(a)

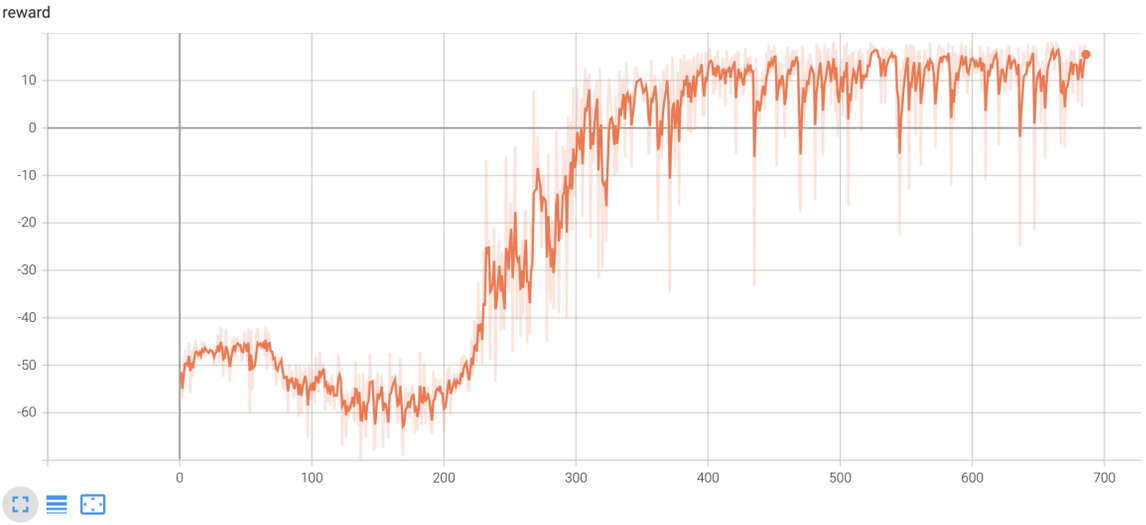


(b)

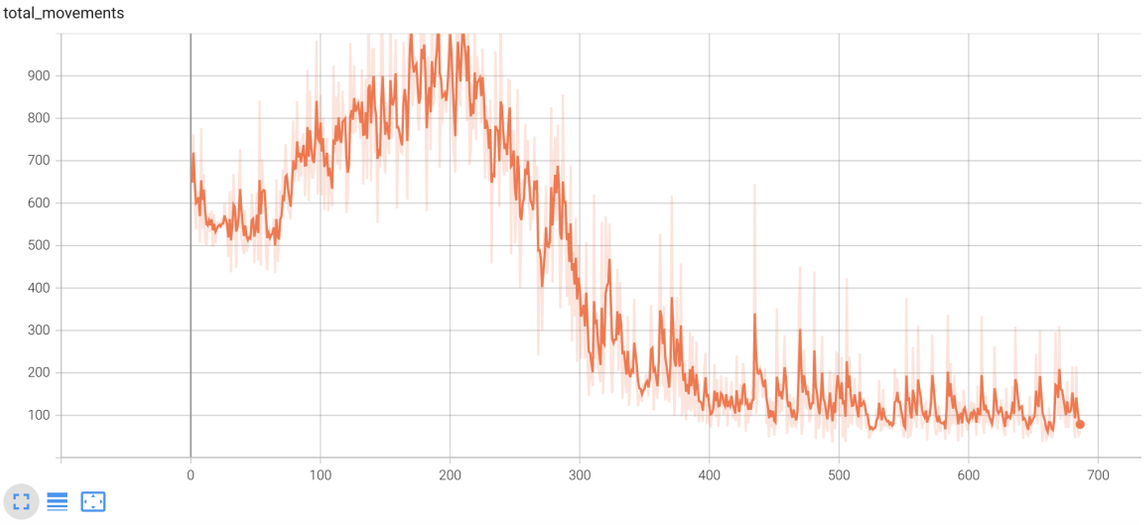
Figure 4: In figure (a) can be seen the rewards obtained by the agent over the training, in figure (b) can be seen the total of movements done by the agent over the training. Each graph has been smoothed with the Exponential Moving Average (EMV) with a parameter of 0.6.

vector reward composed by the score and the movement (-1 if the paddle moved, 0 otherwise). We implemented a classic scalarization method to join the two rewards, giving a different weight to each reward's component. We tried different weights, trying to understand the right scalarization for the two objectives. In the first try we gave a 0.001 weight to the movements and 0.999 weight to the score, reaching the cumulative reward threshold value in 486 episodes with an average number of movements of 1159.87. In the second try we gave a 0.005 weight to the movements and 0.995 weight to the score, it reached the cumulative reward threshold value in 536 episodes with an average number of movements of 748.68. In the third try we gave a 0.01 weight to the movements and 0.99 weight to the score, reaching the cumulative reward threshold value in 427 episodes with an average number of movements of 592.33. In the last try we gave a 0.05 weight to the movements and

0.95 weight to the score, reaching the cumulative reward threshold value in 686 episodes with an average number of movements of 110.32. The results of this last experiment are shown in figure 5. This result is the best that we obtained and shows a reduction in total movements about 91% with respect to the standard agent.



(a)



(b)

Figure 5: In figure (a) can be seen the rewards obtained by the scalarized agent over the training, in figure (b) can be seen the total of movements done by the scalarized agent over the training. Each graph has been smoothed with the Exponential Moving Average (EMV) with a parameter of 0.6. The results refer to the agent with 0.05 weight for the movements and 0.95 weight for the score.

The results obtained are recapped in table 7. We can notice from these results that giving a weight for the movement gives better results with respect to the stan-

dard model, getting an agent that learns to move less. So, the problem scalarization gave the results that we expected, maximizing the score as the standard version but also considering the second objective of minimizing the number of total movements. Furthermore, we noticed that increasing the weight relative to the movements, we got better and better results. The main problem of scalarization still remains, because we needed to try different combination of weights and we can't determine if those that we tried are the best, confirming the trial-and-error approach.

Name	**Score_W**	**Movement_W**	**Episodes**	**Avg. Movements (100)**
Standard	-	-	571	1227.64
Scalariz_1	0.999	0.001	486	1159.87
Scalariz_2	0.995	0.005	536	748.68
Scalariz_3	0.99	0.01	427	592.33
Scalariz_4	**0.95**	**0.05**	**686**	**110.32**

Table 7: Python Experiments final results.

6.2 Julia Results

For what concern Julia experiments we used the same parameters of the Python experiments, as described before in table 6.

What we obtained was that we couldn't train the agent on our machine configuration. This was primarily due to the inability to utilize the GPU with the BANs implementation. As a result, training for just 10 episodes took 15 hours, whereas the average number of episodes required for the Pong game in Python was around 500 episodes. So, later, we tried to use the weights extracted from the convolutional layer of the Python's trained model and loading them into the Flux's convolutional layers, which we then froze. The idea was to do the transfer learning approach, to reduce the number of parameters that needed to be trained from the algorithm. This implementation has been tried not only one the machine that required 15 hours to do 10 episodes (Macbook Pro) but also on two other machines. We didn't get an effective improvement also with this solution. The machine's configurations can be seen in table 8.

Machine Name	**CPU**	**RAM**
Ubuntu computer	i5 8°gen 2.30GHZ 4core	8GB
MacbookPro	Apple M1 Max 3.20GHZ 10 core	32GB
Virtual Machine (given by uni)	Intel Xeon Processor 8core	16GB

Table 8: Specifics of the machines used.

Later, we tried to understand if all the cores were used by the Julia code. We noticed that during the execution Julia uses by default only 1 core, so we tried to use the option –threads=auto and –threads=8 to force Julia to use multiple cores. We noticed, using the htop Linux tool, that the majority of the time the program runs anyway using only one core and rarely uses the other ones. This suggests that the heaviest computational function runs on only one core, while the rest of the program runs on multiple cores. The htop result can be seen in figure 6.

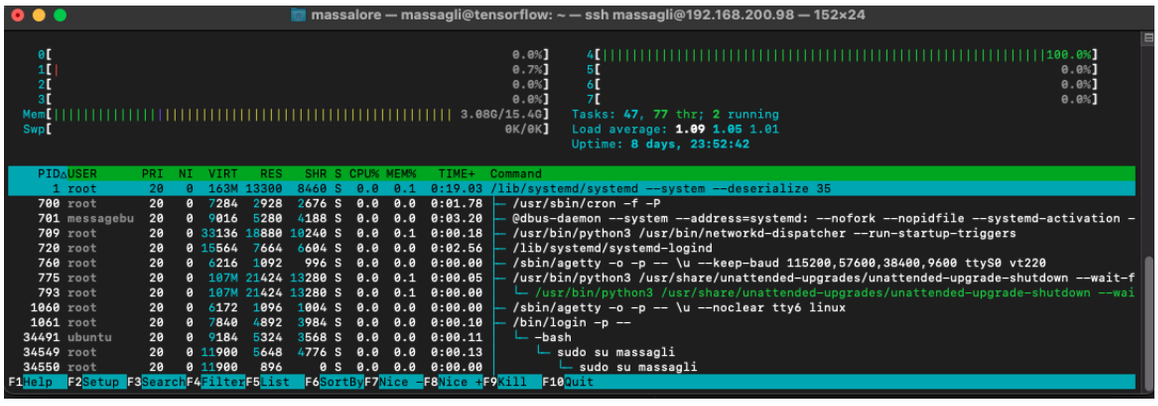


Figure 6: htop Linux tool result.

Then we did the profiling of the code trying to understand what are the bottleneck of the application and the most consuming function. The profiling has been done using the PProf tool integrated in Julia and the perf Linux tool.

Profiling Profiling is a form of dynamic program analysis that measures the complexity of a software program. The goal of profiling a program is to understand the behavior of the program by collecting statistical data about its performance. Profiling is generally used to optimize code. Profiling tasks covers: Performance Bottlenecks, Memory Usage, Function Calls, Concurrency Issues, Input/Output (I/O) Operations.

PProf PProf is a Google tool used for visualization and analysis of profiling data. PProf reads a collection of profiling samples in profile.pb.gz format and generates reports to visualize and help analyze the data. Profiles can be read from a local file, or over http.

Perf Perf is a powerful performance profiling tool that comes with the Linux kernel. It is part of the Linux kernel source code, so it's very closely integrated with the Linux operating system. Perf provides rich and detailed insights into system and application performance, helping developers and system administrators optimize code and troubleshoot performance issues.

We started with doing the profiling with the Perf tool with the related command called perf-record. This command gives in output a perf.data file that can be successively analyzed using "perf-report" command for the command line interface and "firefox profiler" for visualizing it in a web interface. Unfortunately, with this approach we couldn't get insights on the program execution because Julia doesn't insert at compilation time symbols for debugging, so analyzing the program using an external tool results difficult as its not possible to understand in the profile which are the functions and when they are utilized. The interface of the perf results can be seen in figure 7 and figure 8.

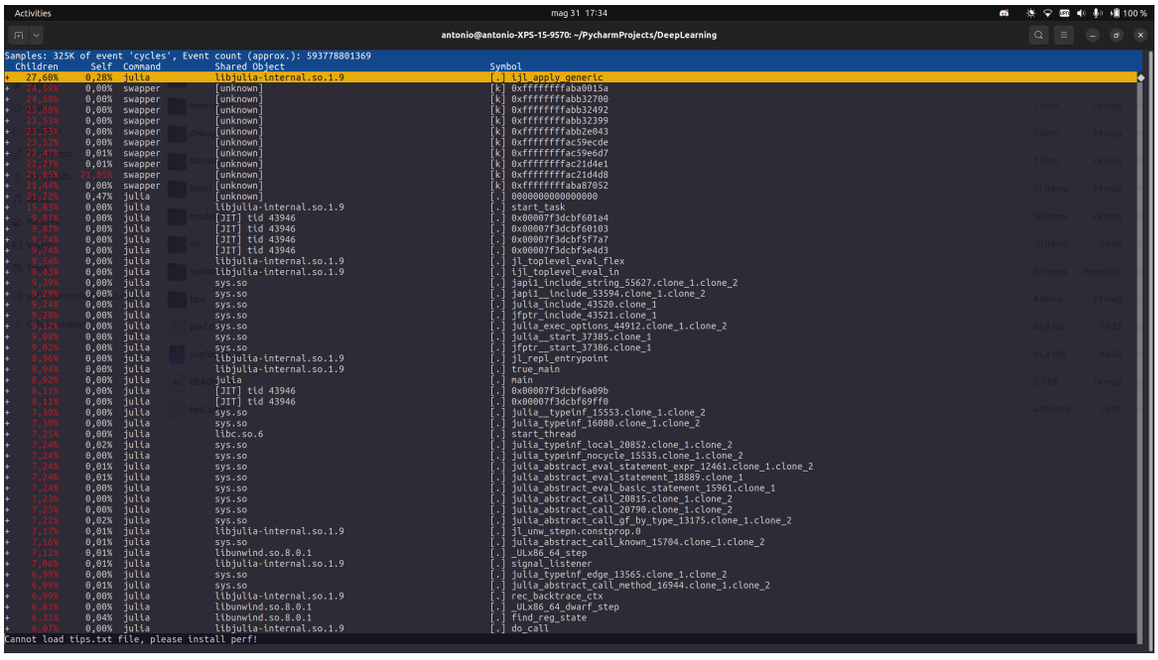


Figure 7: Perf command line profiler results.

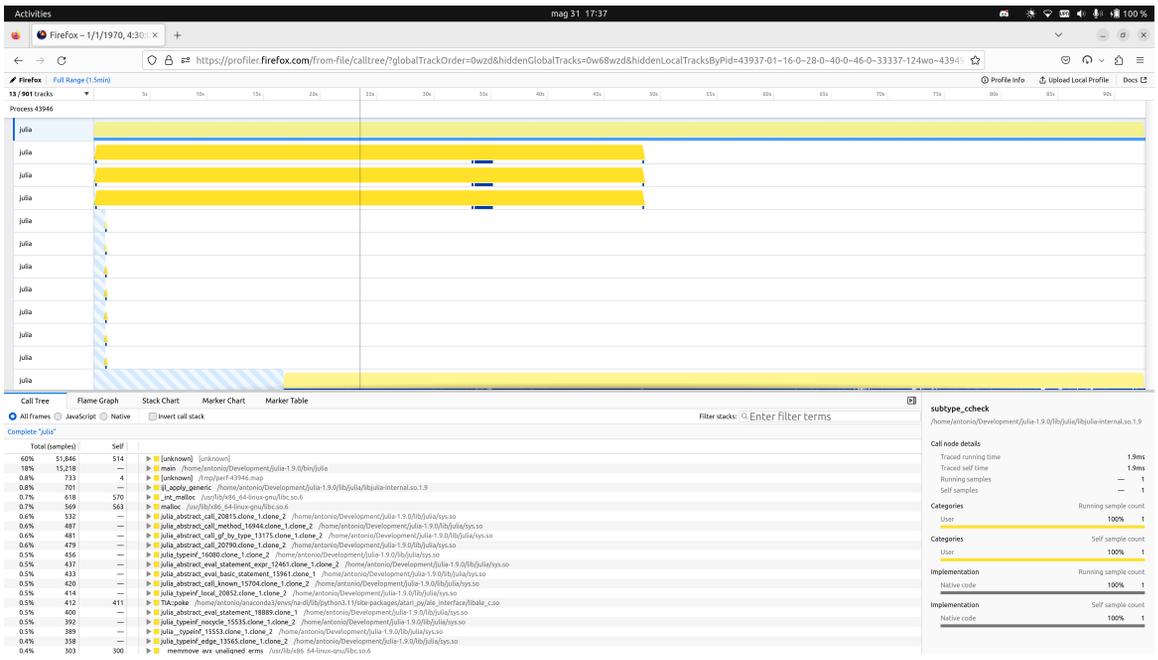


Figure 8: Perf Firefox profiler results.

We then used the PProf tool (integrated in Julia by the relative package). This tool using the profile macro **@profile** let us profile the execution of a function, and calling the pprof() method it opens a web interface to visualize the profiling results. We analyzed the flame graph and, as you can see in the figure 9, we noticed mainly two aspects:

- Half of the computational time is taken by the gradient() function which is used to compute the gradients during the learning phase of the algorithm to update the weights of the network.

- The other half of the computational time is taken by the functions used for the convolution, called **conv_data_direct** and **conv_filter_direct**, contained in the NNlib.jl used by Zygote and Flux.

From these analysis, we then tried two possible approaches. Firstly, we reduced the number of non-standard nodes in the dense layers of the agent's DQN trying to lower the computational cost of updating the weights, because the BAN computations could have been an issue during the training. The model architecture can be seen in table 9, as you can see we lowered the number of nodes from 512

Figure 9: PProf flame graph.

to 64 in both the two dense layers. With this approach we didn't got a relevant improvement, passing from 16s per step to 15s per step, so a reduction of only 1s per step.

Layer Type	**Filter**	**In Channels**	**Out Channels**	**Stride**	**Act. Fun.**
Conv_1	(8,8)	4 (stack_size)	32	4	ReLU
Conv_2	(4,4)	32	64	2	ReLU
Conv_3	(3,3)	64	64	1	ReLU
Dense (BAN)	-	3136	**64**	-	ReLU
Dense (BAN)	-	**64**	numactions	-	-

Table 9: DQN simplified network architecture.

The second approach that we tried, was to reduce the batch size because the operation done in the convolution phase resulted to be the most time consuming. We reduced the batch size using different dimensions, obtaining a lower executing time progressively with the reduction in batch size. This results, obtained using the macro **@time** in Julia, gives us the insight that the convolution computation is the bottleneck, mainly because it is executed on CPU. In figure 10 can be seen the time per step taken by the gradient() with respect to the batch size.

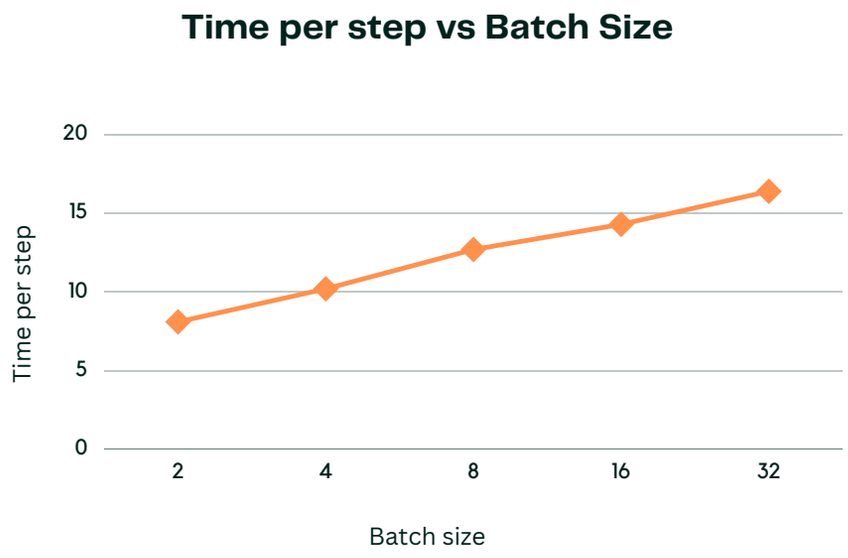


Figure 10: Time per step taken by the gradient() function with respect to the batch size.

Unfortunately, those results still doesn't allow us to perform a complete training analysis due to the fact that it is required approximately 200 hours ($\approx$ 10 days) to complete the training of one model.

7 Conclusions

In this study, we have discussed the Python and Julia implementations of a challenging Reinforcement Learning benchmark, specifically the Pong Atari game. The Python implementation utilized standard numerical methods, employing scalar and vectorial rewards. On the other hand, the Julia implementation explored non-standard numerical approaches, investigating the potential benefits of the innovative Non-Archimedean Reinforcement Learning technique. We achieved promising results in the standard and scalarized Python implementation, with the standard agent solving the environment achieving the cumulative reward threshold of 18.5 in 571 episodes with an average of 1228 movements on 100 episodes, and the scalarized agent (the best) that has solved the environment in 686 episodes with an average of 110 movements on 100 episodes. The scalarized agent shows a reduction in total movements about 91% with respect to the standard one, confirming a positive effect of considering a second objective to be optimized.

In Julia, we attempted to train the agent; however, we encountered computational resource issues. To address this, we employed profiling methods, namely PProf and Perf, to identify potential bottlenecks in our application. Through analysis, we discovered that the computational time required for convolutions was the main obstacle. Despite identifying this issue, training was still not feasible due to the significant computational time when using CPUs instead of GPUs.

The objective of this study is to serve as a valuable starting point for the implementation of Non-Archimedean Reinforcement Learning (NARL) for complex benchmarks (Atari games), offering insights into the analysis conducted and the challenges we encountered. Several potential improvements have been identified:

- **Code adaptation for GPU utilization**: Currently, the BAN numbers cannot be used in GPU computations. Finding a way to modify the code to enable GPU utilization and incorporate the BAN numbers would greatly enhance performance and resolve the computational resource problem.

- **Analysis of Convolution operators in NNlib.jl**: Examining the convolution operators provided by the NNlib.jl library (utilized by Flux and Zygote libraries) and identifying possible issues or areas for improvement to optimize computation speed.

References

[1] Vieri Benci and Mauro Di Nasso. Alpha-theory: An elementary axiomatics for nonstandard analysis. *Expositiones Mathematicae*, 21(4):355–386, 2003.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[3] Mike Innes. Flux: Elegant machine learning with julia. 3(25):602, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[6] G. Silvestri. A new algorithm for lexicographic multi-objective reinforcement learning. Master's thesis, University of Pisa, 2021/2022.

[7] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.