Higher school of economics

Faculty of computer science

Data sciences program

Report for research internship:

Solving the game of Pong using Reinforcement learning

By:

Anwar Ibrahim

Supervised by:

Prof. Sergei Kosnitsov

Moscow 2023

Contents

Introduction:

In this report, I will explain my attempt to solve the Atari game Pong using Reinforcement Learning (RL), this task is one of the tasks required in my thesis, in which I am trying to find an interpretable approach to solve the same game using Reinforcement learning. First, I will briefly introduce the game “Pong”, then introduce the Reinforcement learning concepts and algorithms that I will use to achieve the solution. Second, I will explain a few key points from the implementation, and show the results that I got, since I was able to solve Pong with a moving average reward of almost 15, which is great given the used tools.

Our Task:

Our task is to create an RL agent to decide on the best action to take “how to move the paddle” to reach the score of 15 over 100 consecutive episodes.

Introducing the environment:

Like any other Reinforcement Learning problem we need to have an environment, and similar to the work that will be done in my thesis, I chose the Atari environment “Pong” (Pong, n.d.) that was implemented by the OpenAI library gym (Gym Documentation, n.d.), the environment is about playing the playing ping-pong, the first player to score 21 points is the winner. The following figure shows the game’s frame:

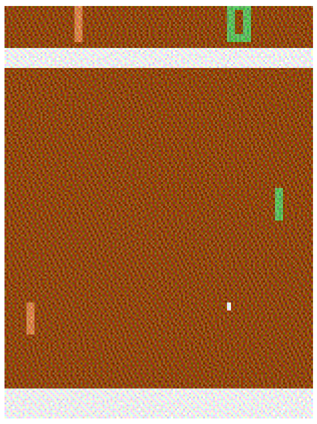


Figure 1:Pong-v4 frame

Pong’s goal:

The agent control’s the right paddle, and it competes against the computer that controls the left paddle, each player tries to keep deflecting the ball away from the goal, and into the opponent’s goal.

Pong’s observation space:

The observation space is how the library returns the game’s state, and for our game by default, the environment returns the RGB image of the shape (210, 160, 3).

Pong’s actions:

Each player in this game can perform any of the actions, from the action space:

Num	**0**	**1**	**2**	**3**	**4**	**5**
action	NOOP	Fire	Right	Left	Right fire	Left fire

Pong’s reward:

The player gets a score of “+1” for getting the ball to pass the opponent’s paddle, and a score of “-1” if the ball passed its paddle.

Episode terminates:

An episode, is several rounds played that end with one of the players winning, and an episode terminates, either if one of the players reaches a 21 score, or if the number of frames passed 400000 frames.

Solved requirement:

Each RL environment is considered solved if the created agent can satisfy, several requirements, and for the game of Pong the created agent, should reach the average score of 17 over 100 consecutive episodes to solve the environment "Pong game"

Reinforcement Learning:

Reinforcement learning is a branch of Machine learning where a model/an agent is created to learn what to do and how to map observations to actions, to achieve the goal of maximizing a numerical reward signal, and the agent learns this decision-making process by interacting with an environment and performing some actions to optimize the resulting feedback from this environment. (Sutton & Barto, 2014,2015) The following figure can explain an RL problem:

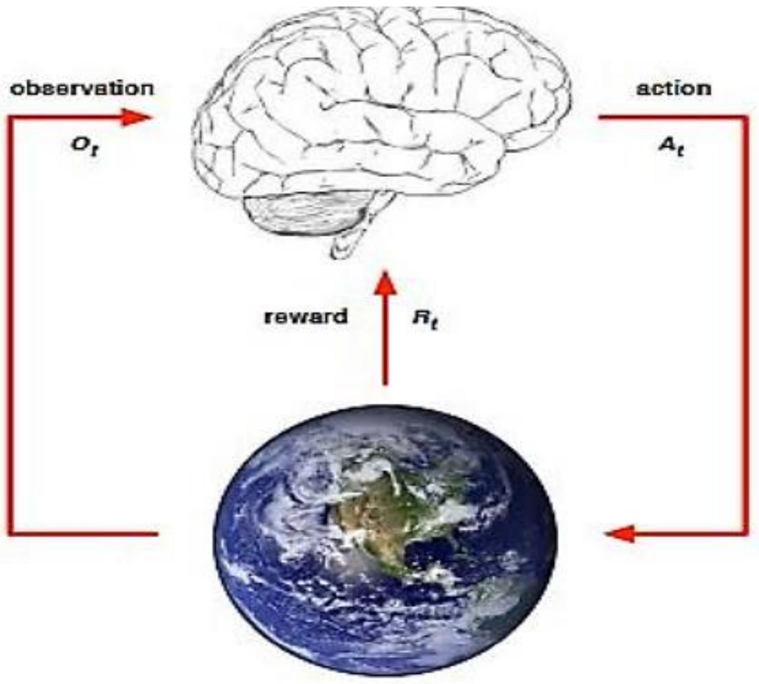


Figure 2: the RL problem is a closed-loop problem that has several elements, the agent here is represented by a brain, and the environment in this figure is represented by the earth, there are also several interactions between these two elements which will be explained in the following chapter.

The RL problem consists of several elements, which are:

- The agent: receives "observations" that explain the environment's status, and according to these observations it chooses and performs the best action, the performed actions will change the environment's status, which will invoke the environment to give the agent a reward and a new observation to describe the new status, the agent will use the reward as a metric to measure the efficiency of the performed action… and this loop will continue until we find the best action for every observation, and this is how we train our agent.
- The environment: will give the agent observations to describe its status, it will also be affected by the agent's actions, leading it to produce rewards that will help train the agent.

To further explain the relationship between the agent and the environment we will show the following figure:

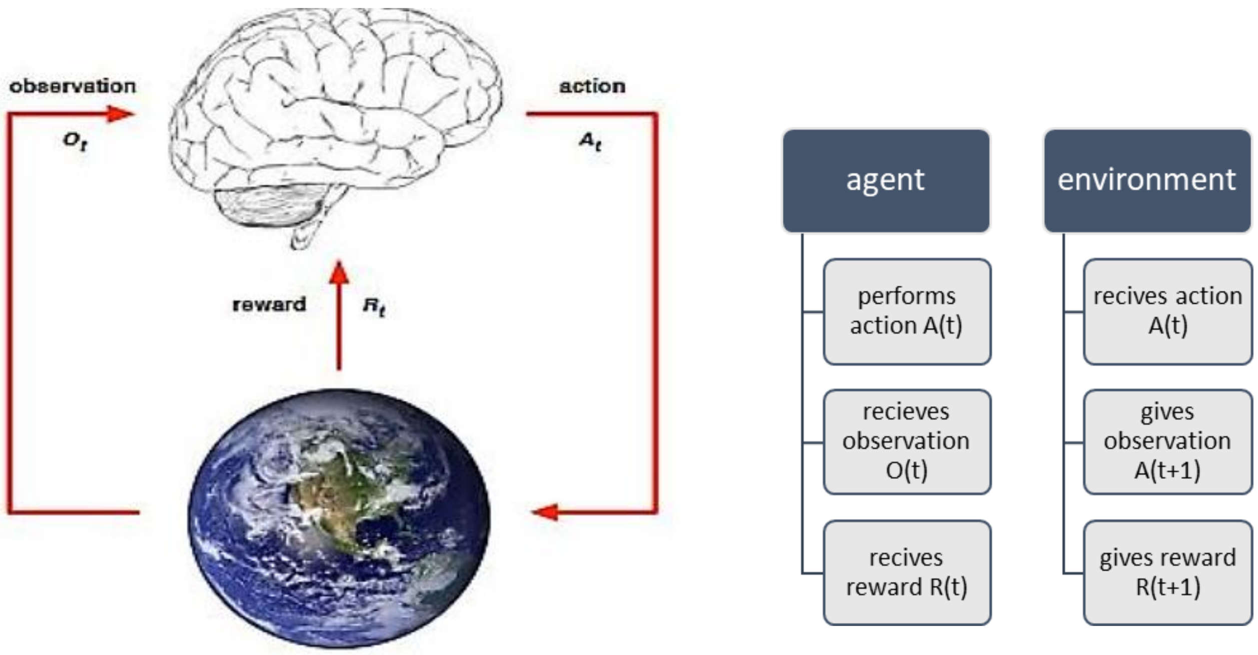


Figure 3: the relationship between the agent and the environment at a time step (t).

We will not go further in explaining RL for Now but for more information, you can see “Using RL to solve Atari games”

Deeper dive into the RL problem:

Our job is to create an agent “the brain in figure 2”, able to understand the input, which can be as simple as an array “like frozen lake game of (Frozen Lake, n.d.) gym” or as complicated as a high-dimensional image, then learns from this input using the previously accumulated experience, to create knowledge representation and reasoning, which will finally enable the agent to create a model of this environment.

Generally, the RL problem is modeled as a Markov Decision Process “MDP”, and an MDP is defined as a tuple, as such:

$$< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$$

$\mathcal{S}$: is the set of states and it could be either a finite or infinite set, and the states can be either discrete or continuous, and a State is a summarization of the chronological record, containing all the useful information, for example when talking about Atari games a state could be the screen frames or the last four frames, and figure 3 shows what we mean by a frame.

$\mathcal{A}$: is the set of available actions, for example, if we are talking about an Atari game like Pong, the available moves are sliding the paddle up or down and you can get more information about this game from the documentation (Pong, n.d.)

$\mathcal{P}$: is the transactions array that shows the probabilities of moving from a state “s” to another one “s'” by performing some action “a”, and the relationship at a timestep “t” is as follows:

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[s_{t+1} = s' | s_t = s, A_t = a]$$

$\mathcal{R}$: is the reward function, and it expresses the reward at a timestep "t" that the environment gives the agent after taking an action "a", given a specific state "s".

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} | s_t = s, A_t = a]$$

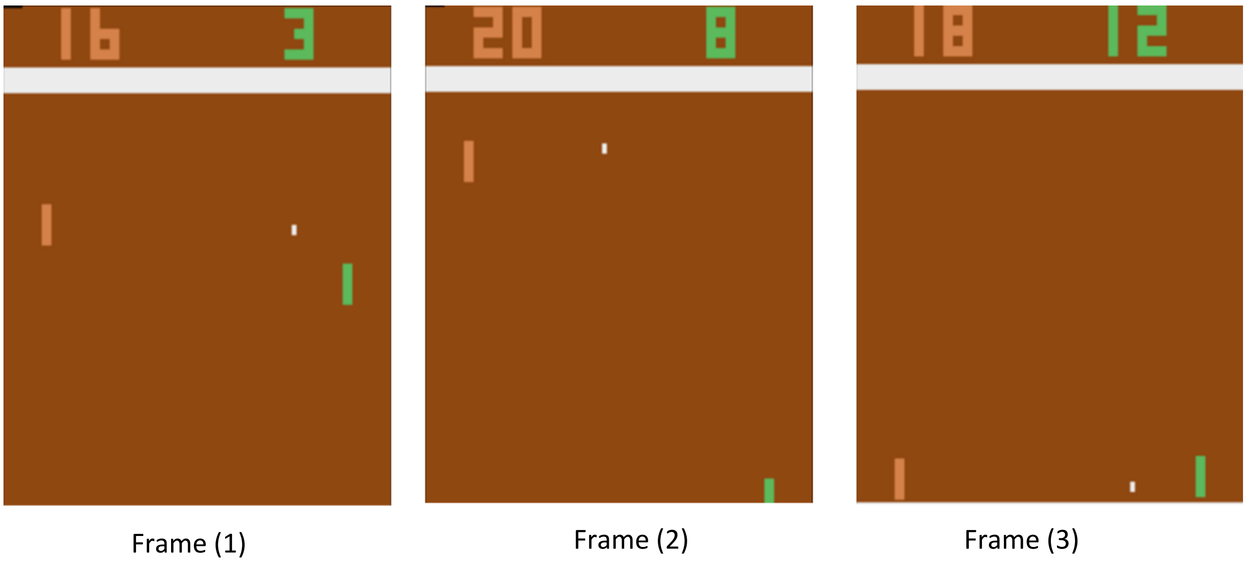


Figure 4: This figure shows examples of different frames of the game Pong of OpenAI gym (Pong, n.d.), where the frame a multi-dimensional image of the game board.

γ: is the discount rate, and it is used to discount the expected future rewards, and its values are in the range [0,1].

On the other hand, the environment could be either a fully observed environment, or a partially observed environment and the following table shows the difference between the two types:

Fully observed environment	**Partially observed environment**
• The agent can see everything in the environment. • The environment state and the agent's state are the same. • This type of problem is a Markov Decision Process (MDP)	• The agent monitors the environment indirectly, for example, the poker player can only see the cards facing up. • The environment state is different from the agent's state. • This problem is a partially observable Markov Decision Process (POMDP)

From the previous explanations, we can notice that the challenge that faces solving the RL problem is that the agent doesn't know the transition function and the reward function, and we need to create an agent to choose the best action for each state to maximize the expected discounted sum of current and future rewards.

Double Deep Q-Learning:

Several RL algorithms can be used to solve our environment, but for our work, we will choose, the Double Deep Q-Learning algorithm because it is the most suitable for our work since our final goal is to achieve interpretability and using this algorithm, we can easily change the structure of the used convolutional neural network with a transformer to achieve interpretability.

And to understand this algorithm we have first to explain a few terms:

Policy $\boldsymbol{\pi(a_t, s_t)}$**:** is a blueprint of the connections between observations and actions in the environment, and it defines the agent's behavior at any point in time.

Model-free RL is when the agent makes decisions without having a model that describes the environment instead, values for different actions are learned directly, through trial-and-error interactions with the environment every time an action is performed.

On-policy learning: is when the agent already has a predefined policy to follow and its only job is to update it as much as it can to reach the optimal one.

Off-policy learning: this is when the agent learns the optimal policy without the help of a predefined policy and behaves using a generic greedy policy of taking actions until it can reach the optimal one for the environment.

Q-Learning is a model-free, off-policy RL algorithm, used to learn the optimal action-selection policy in a finite Markov decision process to maximize the expected value of the total reward over all successive rounds, starting from the current state. This algorithm has a function Q that calculates the quality of the state action combinations:

$$Q: S \times A \to \mathcal{R}$$

Before the learning process starts, Q is initialized to an arbitrary fixed value. Then, at each time step t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} "depending on the chosen action", and Q is updated using the following Bellman equation:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{current value}} \Bigg)}^{\text{temporal difference}}$$

Where:

r_t: is the reward received when moving from the state s_t to the state s_{t+1}.

α: is the learning rate $0 < \alpha \leq 1$, and it determines to what extent newly acquired information overrides old information.

γ: is the discount factor$0 \leq \gamma \leq 1$, and it determines the importance of future rewards.

So, the algorithm is as follows:

Algorithm 1: Q-Learning algorithm

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
   select and act a
   observe reward r and new state s'
   Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] - Q[s,a])
   s = s'
until terminated
```

Deep Q-Learning uses deep Convolutional Neural Networks (CNNs), with layers of tiled convolutional filters to mimic the effect of receptive fields. RL is unstable when using nonlinear functions approximator such as using a neural network to represent Q (Matzliach, Ben-Ga, & Kagan, 2022) and to reduce the instability we use the technique "experience replay".

Experience replay: is a technique to stabilize the training of the q-function by allowing the agent to revisit and learn from past experiences. So, for each time step t all the experiences $< s_t, a_t, r_t, s_{t+1} >$ is stored in a replay memory. Then when training the network, random samples of the replay memory are used instead of the most recent transitions. This will break the similarity of subsequent training samples, which could lead the network to a local minimum (Matiisen, 2015) .

Epsilon- greedy algorithm: is an approach to select the action with the highest estimated reward most of the time. This algorithm aims to balance exploration and exploitation. Exploration allows us to try new actions, while exploitation allows us to use the agent that we trained so far to select the actions. (baeldung, 2022)

Algorithm 2: epsilon greedy algorithm

```
Initialize eps=1 //to guarantee the exploration
Initialize minEps= some small value ex:0.1
Function selectAction(agent, state, eps, actionSpace):
  {
   ran= random value from a distribution (it could be formal or normal)
   If( ran> eps)
              //Exploit the environment by using the trained agent to predict an action
               Action= agent(state)
    Else
              //Explore the environment by selecting a random action from the action space
              Action= random(actionSpace.size)
    Return action
    }
If(eps>= minEps)
               Decay epsilon using a mathematical equation. // new value for eps
Call function selectAction(agent, state, eps, actionSpace) each time we want to get a new
action
```

Double Q-Learning: from the previously defined bellman equation, we can notice that the same Q function is used to decide the best action "highest expected reward", and to estimate the future maximum approximated action-value Q-learning, and this makes it more likely to select overestimated/overoptimistic values. To prevent this problem, we can use two different action-value estimated as: (Hasselt, Guez , & Silver, 2015)

$$\boldsymbol{Q^1_{t+1}(s_t, a_t) \leftarrow Q^1_t(s_t, a_t) + \alpha\,.\,(r_t + \gamma.\,Q^2_t(s_{t+1}, arg\max_a Q^1_t(s_{t+1}, a)) - Q^1_t(s_t, a_t))}$$

Algorithm 3: Double Q-learning algorithm

```
Initialize online-network $Q_\theta$, target-network $Q_{\theta'}$, experience replay buffer $\mathcal{D}$, $C$
// target-network $Q_{\theta'}$is used for action prediction and online-network $Q_\theta$is used for action
//evaluation

For each iteration do:
    For each environment step do:
        Observe the current state $s_t$, and select $a_t$ $using$ $\pi(a_t, s_t)$
        Execute $a_t$ and observe the next state $s_{t+1}$ and reward $r_t = R(s_t, a_t)$
        Store $(s_t, a_t, r_t, s_{t+1})$ in the experience replay buffer $\mathcal{D}$
    For each update step do:
        Sample mini-batch $e_t = (s_t, a_t, r_t, s_{t+1})$from $\mathcal{D}$
        Compute target $Q$ value:
            $Q^*(s_t, a_t) \approx r_t + \gamma . Q_\theta\ (s_{t+1}, arg \max_{a'} Q_{\theta'}(s_{t+1}, a'))$
         Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$ // we want to
minimize
         //the mean squared error between $(Q_\theta, Q^*)$
         Update target-network parameters every $C$ steps   $\theta' \leftarrow \theta$
```

Deep Double-q-Learning is practically the same algorithm but it uses a deep multi-layered neural network as an architecture for the online network, and the target network.

Our implementation for the solution:

In the following section, I will explain a few key points of the implementation part, the full implementation is attached as a PDF file and a python file that can be run using google Colab, the attached code has comments to explain each step.

The input:

From our environment's frames "pong-v4", we can see that the observations lack a lot of important information like the ball's direction, it also has useless pixels which will only make our computations more expensive without yielding any benefits, so it is important pre-processing the frames to make them more beneficial:

Function 1: preProcess(frame)

```
Convert image to grayscale
Crop useless pixels
Resize the image// to make it rectangular again after cropping useless pixels
Normalize the image //less computationally expensive
```

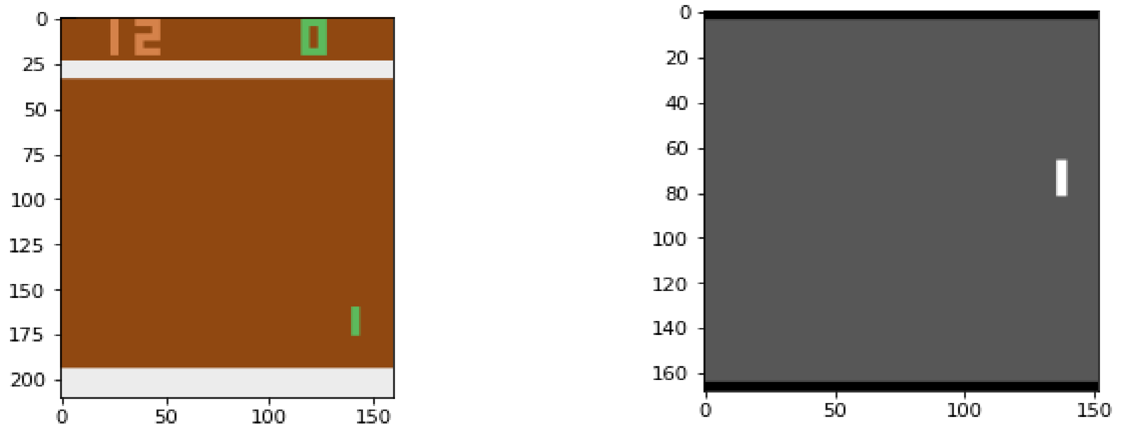


Figure 5:the figure on the left shows the observations before preprocessing, and the one on the right shows the result after preprocessing.

To include the information about the ball’s direction, we will take four consecutive frames instead of one, and this will be the agent’s input, the following figure shows the new state:

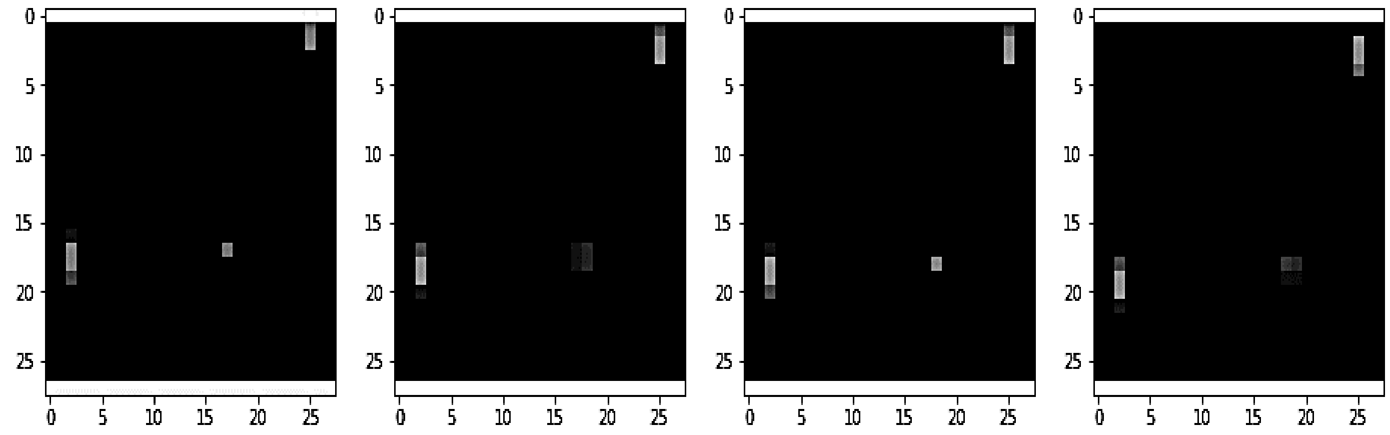
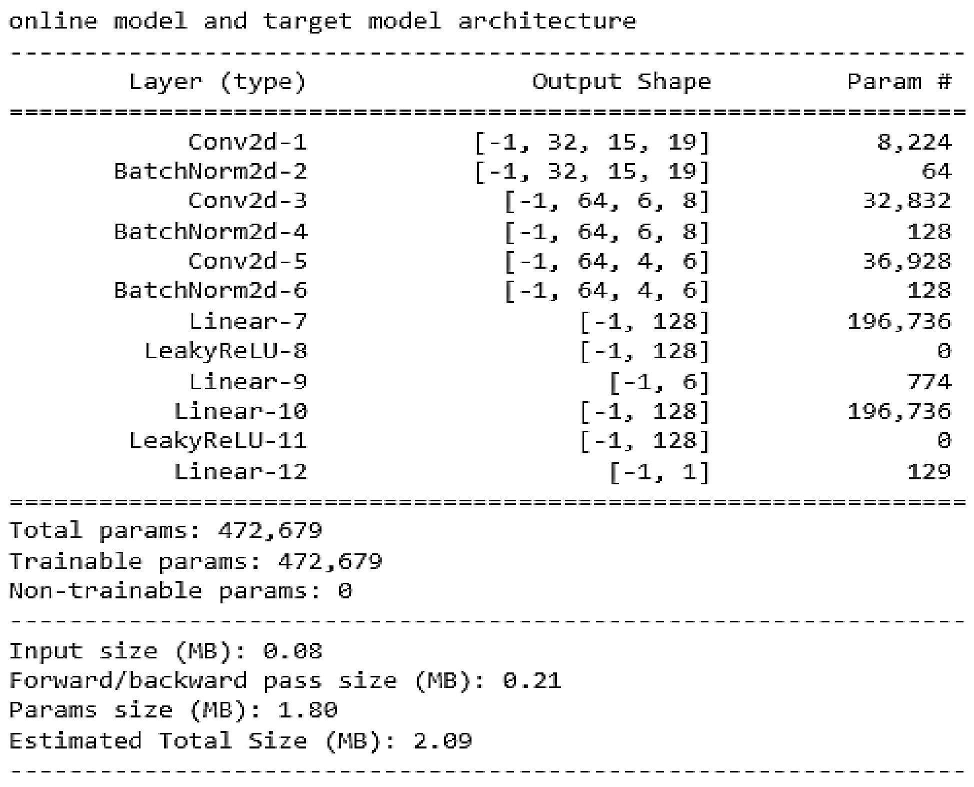


Figure 6: Our Relational Agent's input state.

The CNNs architecture:

We have tried three architectures to solve this environment, but the first two rendered unsatisfactory results, so in this report, I will only include the architecture with the best results:

According to the Deep Double Q-Learning algorithm, we already know that we should have two neural networks, one to predict actions “the online model in our implementation”, and the other will be responsible for evaluating the future action-values “the target model in our implementation”. Both two neural networks have the same structure, which can be shown in the figure below:



```
online model and target model architecture
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 15, 19]           8,224
       BatchNorm2d-2           [-1, 32, 15, 19]              64
            Conv2d-3             [-1, 64, 6, 8]          32,832
       BatchNorm2d-4             [-1, 64, 6, 8]             128
            Conv2d-5             [-1, 64, 4, 6]          36,928
       BatchNorm2d-6             [-1, 64, 4, 6]             128
            Linear-7                  [-1, 128]         196,736
         LeakyReLU-8                  [-1, 128]               0
            Linear-9                    [-1, 6]             774
           Linear-10                  [-1, 128]         196,736
        LeakyReLU-11                  [-1, 128]               0
           Linear-12                    [-1, 1]             129
================================================================
Total params: 472,679
Trainable params: 472,679
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.08
Forward/backward pass size (MB): 0.21
Params size (MB): 1.80
Estimated Total Size (MB): 2.09
----------------------------------------------------------------
```

Figure 7:: architecture of online model and target model.

The training step:

We trained our model on google colab GPU for 750 episodes, with a learning rate of 0.00025, we had a lot of RAM crashes, which made the training process more challenging, because even though our CNN architecture was not that complicated, we were storing 2 CNNs at each step in addition to the experience replay memory and those parameters are big and that affected our results because we need a more complicated CNN architecture to overcome the saturations in the results.

To overcome losing our progress after every ram crash we saved our model, and output results to google drive, and that was made clear in the attached code.

The results:

The following figure, plots the moving average of the rewards over windows of the size 100, and we can easily see that our model had reached a running average reward of almost 15:

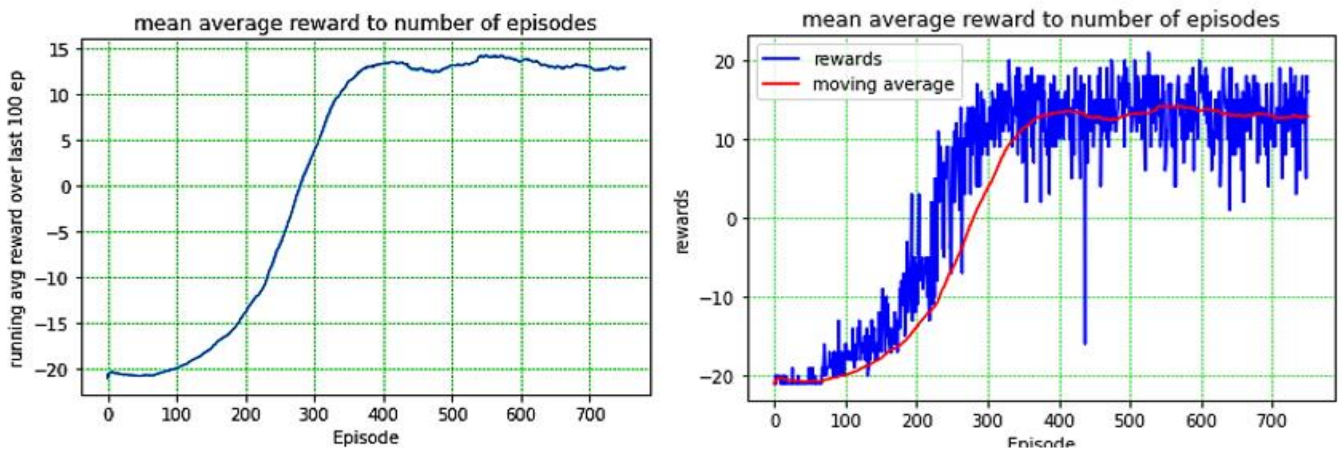


Figure 8: the figure to the left shows the running average of the reward over 100 episodes windows. the figure to the right shows both the rewards and the running average over 100 episodes windows.

We can notice that the running average gets saturated at this value, and this can be because the used neural network is not big enough using a more advanced CNN structure will defiantly yield better results.

The following figure duration-episode also shows the duration that each episode took to

terminate is getting better:

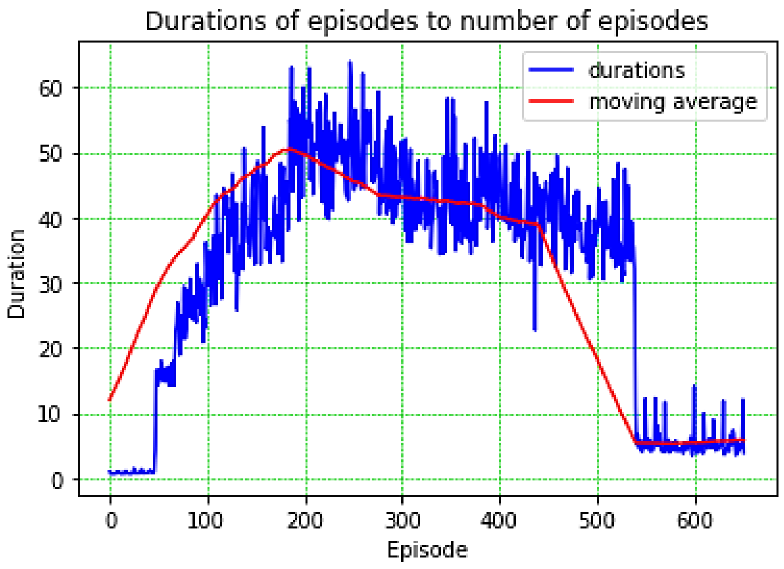


Figure 9: shows how the episode termination duration changes as the model train.

We can see from the following figure that our model is achieving the same good results but at a better time, so this means the model is improving, however, it saturates after almost 550 episodes, and it is for the same reason “we need a more complex model to achieve better results”…

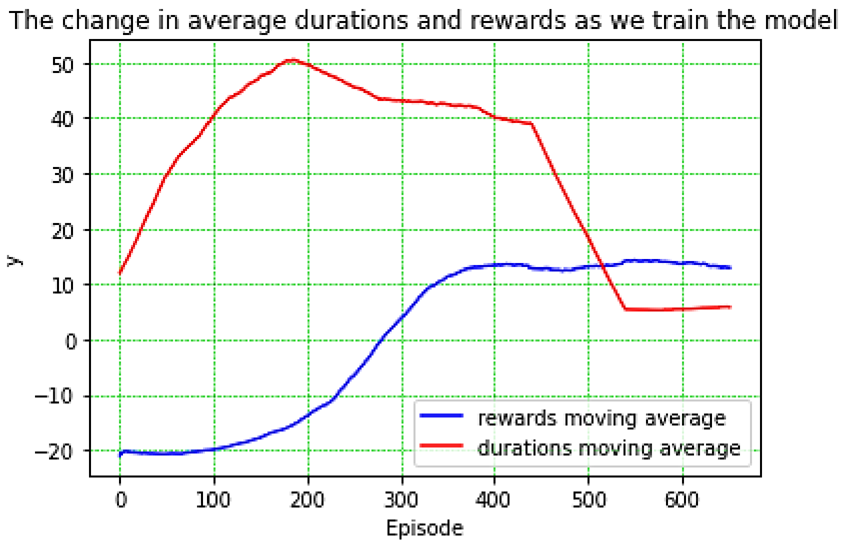


Figure 10: the change in average durations and average rewards as the model trains.

Conclusion:

In this report we solved the task of creating a model to solve the game of Pong using a Double deep Q-learning algorithm, I managed to reach an almost 15 running average reward over windows of the size 100. We will use the same algorithm to solve the thises task, but with different models structure for the online, target model, and the new architecture will be “vision transformer (vit) or, decision transformer” since we want to add the concept of interpretability and the transformers train an attention matrix with the model and this matrix can be used to show which parts of the input our model is paying more attention to. The training process to achieve these results encountered problems with the RAM since we are storing two CNNs with almost 500000 parameters each and a replay of the size of at least 40000 and storing all these parameters in the cash led to a lot of crashes, fortunately, the obtained results were good even though we had this problem, however, to solve the model with “vit” it will take a lot more parameters so this problem will only get worse but hopefully we will get a better machine for the future work and we can overcome this problem.

References

[1] baeldung. (2022, 11 11). *Epsilon-Greedy Q-learning*. Retrieved from baeldung: https://www.baeldung.com/cs/epsilon-greedy-q-learning#:~:text=The%20epsilon-greedy%20approach%20selects,what%20we%20have%20already%20learned.

[2] *Frozen Lake*. (n.d.). Retrieved from OpenAI: https://www.gymlibrary.dev/environments/toy_text/frozen_lake/?highlight=frozen+lake

[3] *Gym Documentation*. (n.d.). Retrieved from OpenAI gym: https://www.gymlibrary.dev

[4] Hasselt, H. v., Guez , A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. *cs.LG.* Google DeepMind.

[5] Matiisen, T. (2015, December 19). *DEMYSTIFYING DEEP REINFORCEMENT LEARNING*. Retrieved from COMPUTATIONAL NEUROSCIENCE LAB: https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

[6] Matzliach, B., Ben-Ga, I., & Kagan, E. (2022). Detection of Static and Mobile Targets by an Autonomous Agent with Deep Q-Learning Abilities. *entropy*.

[7] *Pong*. (n.d.). Retrieved from OpenAI: https://www.gymlibrary.dev/environments/atari/pong/?highlight=pong

[8] Sutton, R. S., & Barto, A. G. (2014,2015). *Reinforcement Learning: An introduction.* London, England: The MIT Press.