RL-PONG : Playing Pong from Pixels

Aakriti Adhikari and Yang Ren
CSCE 790-001
Project Proposal
CSCE 790-001: Deep Reinforcement Learning and Search
Date: 12/6/2021

Index Terms—**RL agent, pong, DQN, DDQN**

***Abstract*—This paper demonstrates RL pong, a RL agent that can play pong against a simple AI agent. Our objective is to utilize the learning obtained throughout the course to build an agent capable of learning pong to beat the opponent. To do so, our work implements two algorithms: Deep Q-Networks (DQN) and Double DQN (DDQN). Our evaluation the result shows that our agent can win against trained agent with a win of 62% using DQN and 82% using double DQN. Also, our agent has been able to learn the strategy to pass the ball effectively. Further, in our work, we overcome the computation complexity by adding a fewer steps in pre-processing like, transforming an image to a tensor object, changing update frequency, memory buffer size etc. Such an approach along with hyper parameter tuning is useful in exploiting the functionalities of CUDA cores of GPUs and speeds up the training. Our training improve by 8 times than without transforming to tensor object.**

I. Introduction

A. Overview

Reinforcement learning (RL) in Artificial Intelligence (AI) is suitable for learning to play and improving in playing the game over time. With the success of ATARI where AI agent was able to beat humans by learning from raw pixels, RL is being applied in many other games. By the combination of RL and deep learning, commonly known as Deep Reinforcement Learning (DRL), the performance of RL agents is improving and it can discover hidden patterns in the game. Our project RL pong implements DRL to understand the working of RL agent to play pong by implementing two algorithms: (1) Deep Q-Network (DQN) with replay and (2) DOuble DQN with replay. The project work aims to demonstrate the implementation of an RL agent called RL-Pong (which we have named AAR agent). Our AAR agent plays the game of Pong from pixels data.

We implement DQN and Double-DQN by training our agent using a deep learning approximator with two-layered convolutional neural networks (CNNs). The game environment is based on Open AI's gym environment [1]. The environment has been modified by [10]. Our environment is based on [10], where a simple agent has been set as an opponent. This opponent is capable of following the ball on the y-axes of the screen. Unlike a simple AI agent, our agent cannot access environment variables like coordinates of the ball in the field and needs to entirely learn to play from pixels. In our environment, there are two players and they can control two paddles at opposing sides of the table. The players must choose between three actions: move up, move down or stay in the same place. The goal of the game of our agent is to keep the ball in the game by passing it back to the opponent's side of the table and implement a strategy to score more by making the opponent miss the ball. The game is played as episodes. The episode ends when one of the players misses the ball. When an agent scores, it receives a reward of +10 at the end of the episode, and if the opponent scores then it receives -10. We measure the performance through reward during recent episodes and win rate. We explain our architecture and approach in detail in the next sections. We train and test our agent to compete with a pre-trained AI agent called Simple AI agent.

B. Challenge and Observation

The major challenge throughout the process is to tackle with **the computational complexity**. Even with GPU NVIDIA GTX1050, our network was extremely slow. It required 40 mins to complete 100 episodes. This halted our progress to great extent. Further, being new to the torch environment, optimizing tensor allocation was the greater challenge.

This led to a realization that we spend the majority of time on improving hyper-parameters but adding two simple steps in our preprocessing improved the computational speed by 8 times. We transform an image into a tensor object and tune our target update to every 50 frames (compared to 2500 frames). It took 5 mins to complete 100 episodes in GPU NVIDIA GTX1050. Note that the comparison is about the initial episodes of training. When network starts learn, it consumers more time depending upon how long it can bounce the ball back. Further, we tune several hyper parameters to improve the performance.

Our final result shows that our agent can beat opponent 82% of time with DDQN and 62% of time with DQN. Further, DQN plays around 13 games in 5 minutes whereas DDQN plays 15 games in 5 minutes. Our observations show that our agent learns to tackle an opponent by returning the ball with a wider angle and bouncing the ball and in DDQN, it can also develop strategy to beat its opponent. Our comparison results demonstrate that DDQN is effective than DQN to beat opponent and develop strategy.

Due to time limitations and the debugging issue, we couldn't solve Proximal Policy Optimization (PPO) as listed in our initial goal for the project.

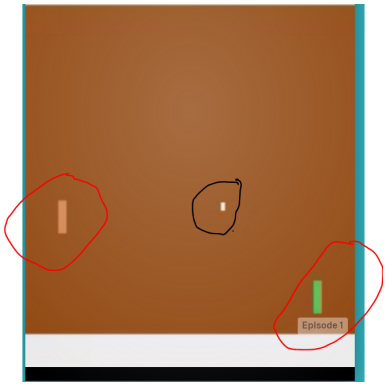

Fig. 1. Open AI Gym environment for pong

II. Related Work and Background

A. OpenAI gym (Pong-v0)

OpenAI Gym is an open-source toolkit for studying and comparing reinforcement learning-related algorithms, containing many classical simulation environments and various data [2]. The current research in reinforcement learning faces the lack of standardization of the environment used, making it challenging to replicate published results and compare results from different papers. The gym provides an excellent solution to this problem. OpenAI Gym provides interfaces to many problems and environments (or games) that users can use for testing and simulation by simply calling them without knowing much about the internal implementation of the game.

B. Pytorch

PyTorch is a well-known deep learning framework that supports GPU acceleration and automatic derivation and has received a lot of interest from the academic community in recent years [5]. Pytorch library includes different algorithms for building and optimizing convectional neural networks (CNNs), especially for image recognition. Therefore, we decided to use pytorch as the basis for our Deep Q Network (DQN).

C. RL course-DQN agent

In RL course, in our coding homework 2, we learn to build DQN using experience replay. The course motivated to build framework for DQN in our work. In the previous studies, DQN could apply to the different games in the Atari 2600 platform and perform as professional human players with minimal a priori knowledge by optimizing the algorithm to learn successful policies directly from high-dimensional sensory inputs [4].

D. Weight initialisation

The weight initialization process entails initializing each node's weights and biases before training the deep neural network [5]. The weight initialization is crucial to whether the network could get good results or how fast it converges. Sometimes, even good training results are not obtained because of the parameter initialization.

E. Other Sources

To achieve better performance of the DQN model, hyperparameter tuning is a common method to find the optimal parameter setting and improve the performance from the baseline model [6,7,8].

III. Approach

Our work has two motives: (a) to compare two approaches: DQN and Double DQN, and (b) to understand the real-life application of RL topics discussed in-class lectures. Our approach is similar to the original Atari paper [9] but differs in the loss function, optimizer, and the utilization of a target network to stabilize the training. For the rest of our hyper parameters, we follow [9] due to the higher performance equivalent to the expert human player is achieved. We discuss our approach below:

A. Data pre-processing

The feedback from the Pong environment is a raw colored image of size 200x200. We did data pre-processing in three stages:

- We first change the color into grayscale then to binary. This assisted network to distinguish elements from the background for better convergence.
- We down sample each frame to 100x100 image.
- We transform an image into a tensor object. This is essential when working with images and CUDA cores of GPU.
- We stack four adjacent observed frames from the environment into a stacked array of (4,100,100).

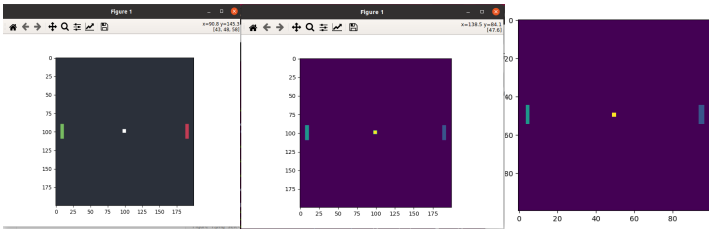

Fig. 2. (a)Original image; (b) grayscale image; (c) final image

In the process of stacking, we tried following the original DQN atari paper where they stacked 4 frames together. We also worked with 2 and 3 frames. However, we observed that with frames lesser than 4, it is not able to learn the movement. We provide these stacks via a buffer implemented into the experience replay memory. We are inspired by other relevant

works that use a buffer of 4 to provide stacks. The process is :

- Time t $>$ 4: The environment returns a transition at time t and pushes it to buffer. If a buffer is full with 4 images, the observations are stacked and a new transition is composed by the stacked frames, the most recent stacked frames, action, the reward is pushed into replay memory. To take an action, the agent looks into the buffer and retrieves all the observations in the past transition, and stacks them. Before observing the transition at time t+1, the oldest transition in the buffer is deleted.
- Time t $<$ 4: The environment returns transition at time t and pushes it to the buffer. To take an action, the AAR agent retrieves all the observation in past transition, stack them, and repeat the most recent observation till the sequence length is 4.

B. Deep Q-learning network (DQN)

In our project, we select two Q-learning algorithms: DQN and DDQN. In each state s we observe the game and try to estimate the Q-values.

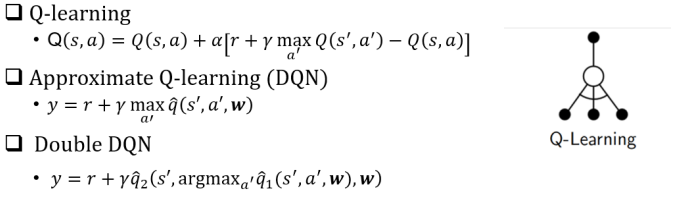

Fig. 3. Equations for DQN and DDQN

From the equation, we observe that when in *state s*, we take *action a* to reach to *new state s'* with *reward r* and then, we want to update our estimate of future rewards while being in *state s* and taking *action a*.

DQN has many variations to remove **maximization bias**. One such DQN that removes maximization bias via two Q-value the estimator is double DQN. Here, we use model Q and target model Q' to use Q for action evaluation and Q' for action selection. In DQN, we maximize over all the Q-values over all the possible action but in double DQN we estimate the value of chosen action instead. The action chosen is the one selected by our policy model.

In our approach, we apply neural networks as function approximators. We use two networks called policy network (Q) and target network (Q'), which are also updated separately. The policy network predicts the next state for all possible actions and the target network predicts the next state values for all possible actions and then selects the action that maximizes the next state value. We implemented Huber loss between the max value of target network output and policy network's prediction with the action selected during the game. The policy network Q is optimized with an Adam optimizer. The way we

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Fig. 4. Deep Q-Learning

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe 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 replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Fig. 5. Double Deep Q-Learning

write our algorithm follows Figure 4 for DQN and Figure 5 for DDQN.

Our neural network structure is discussed in the subsection below and is based on parameters literature for Atari games [9].

C. Convolutional Neural Network

Our network needs to learn from pixels. To process such low-level data like images, deep learning algorithms have proven to yield better performance. We apply CNN as a neural network for function approximators. Our CNN architecture is shown in Figure 6 and the table below.

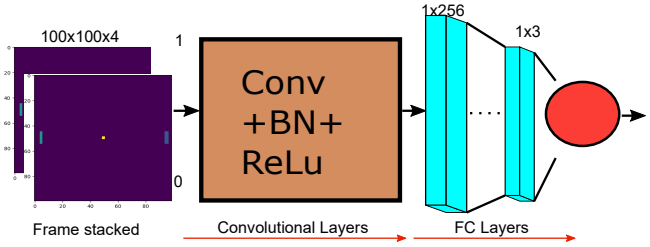

Fig. 6. Our CNN system flow.

Table: Conv+BN+ReLU parameters

Layer	Channels	Filter size
Layer	1	
Conv2d	16	8 (stride 4)
Layer	2	
Conv2d	32	4 (stride 2)
Layer	3	
Flatten		
Layer	4	
FC	256	

D. Experience replay

To remove **catastrophic forgetting** problem in DQN, we store tuples of (s,a,r,s') in replay memory. Then, we draw batches of data from replay memory to draw previous experiences.

E. Weight initialisation

In our work, we experiment with different weight initialization techniques: random initialization, normal initialization and Xavier uniform distribution and found better results with normal initialization with mean 0 and scale 1. Further, pytorch clip of -0.1 and 0.1 was kept for entire training for better convergence.

F. Hyper parameter tuning

In our project, optimizing hyper parameters is a bit of a challenge even though existing pieces of literature are available. This is mainly due to computational limitation and code optimization for greater speed. We have these hyper parameters to tune: replay memory, optimizer, target update frequency, and the number of frame stacks. In the table below we show value for our old hyper parameters and new ones:

Table: Hyper parameters

Hyper parameters	**Old**	**New**
Replay buffer size	100,000	40,000
Batch size	32	128
Discount factor	0.99	0.99
Learning rate	1e-3	1e-4
Optimizer	RMS prop	Adam
Frame stacks	4	4
Target update	2500	50

The exploration in our network follows greedy-epsilon with exploration probability that decreases gradually from 1 to 0.5 till 10,000 episodes with minimum epsilon of 0.1. Earlier, we set this to decrease from 1 to 0.1 till 10,000 episodes. This helps our agent to first explore more and learn new experiences and gradually, decreases exploration to improve exploitation. Such strategy is called **Greedy in the Limit with Infinite Exploration (GLIE)**.

G. Training Strategy

We train our DQN and DDQN with GLIE strategy. We train agent to play with simpleAI for nearly 10,000 games. The models are saved periodically in order to test against simpleAI.

IV. Experimental Results

A. Training results

In figure below, we show out training results for approx 10,000 episodes. Our epsilon decreases from 1 to 0.5 and further decreases. Due to time constraint, we are only able to run till these episodes (we discovered this method very late in our project).We believe that the better performance can be observed by decreasing exploration to 0.1. It is important to notice that the /textbfwin-rate is at around 0.25 (1 being the highest) during training due to our epsilon greedy being very high, thus prioritising exploration, and the score shown being an average of 100 games. **Our previous win rate on training was 0.00142 (project progress) and 0.0091 (during project presentation)**

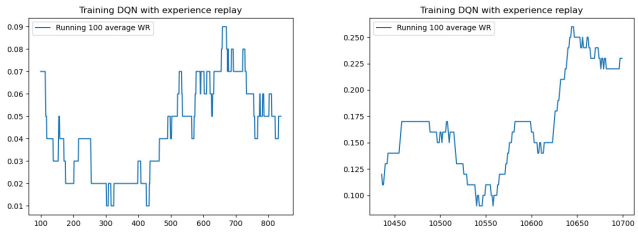

Fig. 7. Training curve for DQN (note: it broke down on one of our desktop due to cuda error so we needed to restart the process time and again for this part so only two figures are attached here

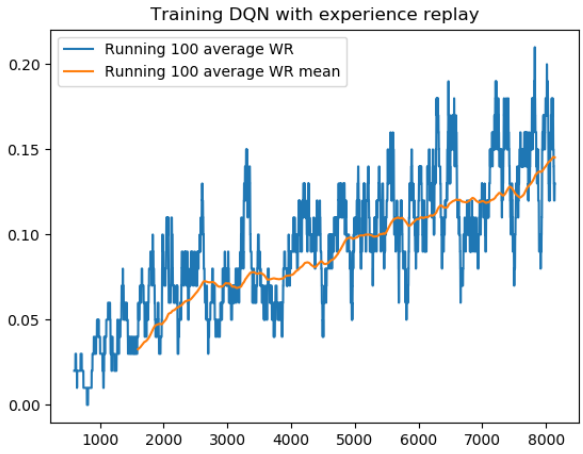

Fig. 8. Training curve for DDQN (note: it was trained for 8000 episodes and it still consumes larger amount of time than DQN

Observations about algorithms in training:

- Double DQN takes larger amount of time for each episodes than DQN.
- In Figure fig:ddqntime, we can see that double dqn plays episodes for longer amount of time. It can probably because even though agent is not winning, it is able to strike back for longer duration.

- In double DQN, agent is able to learn some strategies to play for longer duration than DQN , probably due to no maximization bias (see Fig 10)

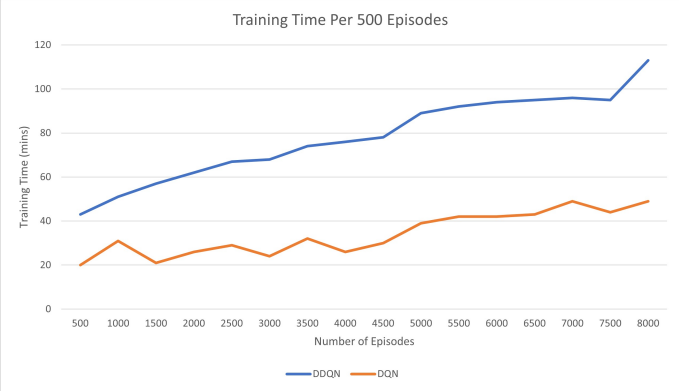

Fig. 9. Time per 500 episodes for DQN vs DDQN during training

Observations about AAR agent in training:

- AAR agent starts playing for longer duration after certain episodes and can follow the ball i.e it can make sense about the direction from where ball is coming and starts to move towards that direction.
- AAR agent cannot take longer coverage due to which it misses out some the ball even if it can sense the direction.
- AAR agent learns to throw ball in direction opposite to opponent but hasn't converged to policy to beat the opponent. In DDQN, it has learned a move to throw ball such that opponent takes longer time to reach towards the ball as shown in Figure 10.This is missing in DQN.

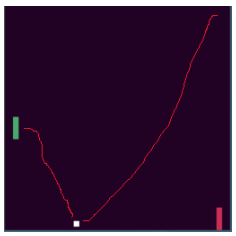

Fig. 10. DDQN AAR agent throwing ball at longer range

B. Testing results

Please see our video: We have attached our testing outcome in the form of short video for both DQN and DDQN in our submission folder (inside output_video) Our results shows that during testing, AAR agent is able to beat its opponent. For DQN, it beats around 62% for 157 games in 60 minutes while for DDQN, it beats around 82% of time for 182 games for DDQN. The timespan for 182 games is 60 minutes. In Figure 11 and 12, we see the win percentage and total number of episodes in 60 minutes.

Observations about algorithms in testing:

- *Time:* It can be seen from the Figure 12 that DQN plays for small number of episodes in comparison to DDQN.

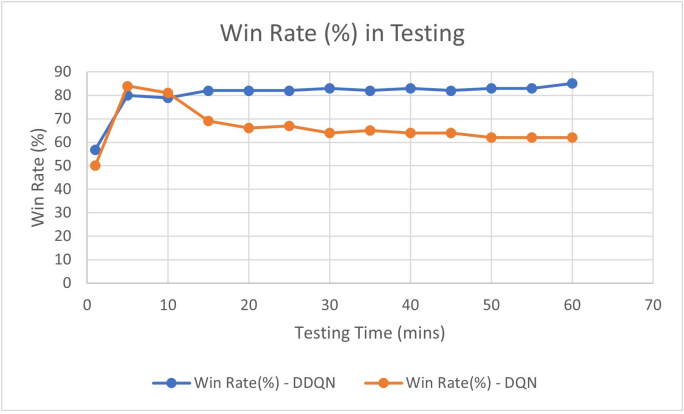

Fig. 11. DQN and DDQN win within 1 hour of game

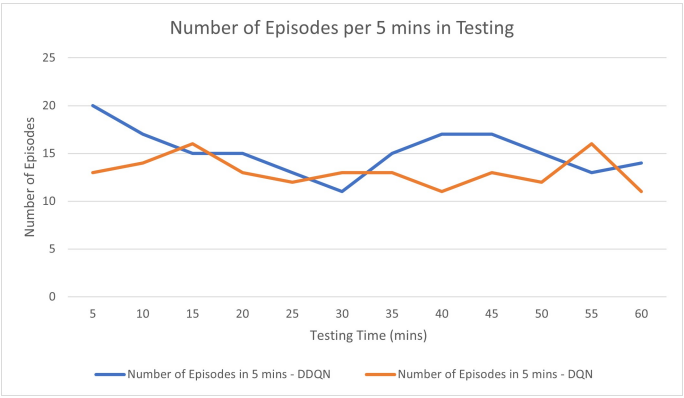

Fig. 12. Number of frames taken to play 5 mins of game

From testing video we can see that DQN initially bounces back well but couldn't develop strategy to beat opponent so keeps playing for longer duration. But, DDQN quickly beats opponent and moves to play next episodes.

- *Strategy:* Double DQN can play game for longer run as well as know how to beat the opponent by passing ball in such a way that opponent will miss it. But, in DQN, it can pass ball back to opponent and plays for longer duration but it has not still developed strategy to beat opponent after games. When you see the video, you see that it passes only and passes ball to convenient direction rather than difficult direction for opponent.

Observations about AAR agent in testing:

- AAR agent in testing can make sense of direction.
- AAR agent is quick and scrolls up and down quickly to bounce the ball to opponent.
- AAR agent can play pong for longer duration. For DQN, it plays **13 games in 5 minutes** and for DDQN, it plays approx **15 games in 5 minutes**.
- In DDQN, AAR agent also strategies to beat the opponent.

V. Conclusion

Deep Q-Networks (DQNs) is a well-established reinforcement learning method for training an agent to play Pong. DQNs allow us to build upon Q-learning by using a deep neural network as the Q function approximator and applying gradient descent to minimize the objective function. Following the Double DQN approach as shown in Figure 4 and 5 to teach an agent to play various Atari games, we are able to implement DQN to teach an agent to play Pong. Even though our AAR agent learns a few moves to tackle the opponent and play pong with a significant win rate, it needs strong optimization to improve its performance for DQN. With a win percent of 82% for DQN and 62% for DDQN, we observe that in this particular environment having a double DQN performs the best to win against the opponent.

A. Improvements for future

In our work, we train our AAR agent to compete against simple AI. However, training can be done in a hybrid approach where the agent can play one game against an opponent and one game against itself. The research works suggested that such a hybrid method can improve the win rate significantly. We leave this to our future work.

In addition to having a hybrid approach, comparing the existing work with policy gradient method can provide insight into the shortcomings of DQN as an approximator.

References

1) Gym-OpenAi, "Pong-Vo," 2021 [online]. Available at : https://gym.openai.com/envs/Pong-v0/
2) Brockman, Greg, et al. "Openai gym." arXiv preprint arXiv:1606.01540 (2016).
3) Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019): 8026-8037.
4) Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." nature 518.7540 (2015): 529-533.
5) Cruz Jr, Gabriel V., Yunshu Du, and Matthew E. Taylor. "Pre-training neural networks with human demonstrations for deep reinforcement learning." arXiv preprint arXiv:1709.04083 (2017).
6) Sebastianelli, Alessandro, et al. "A Deep Q-Learning based approach applied to the Snake game." 2021 29th Mediterranean Conference on Control and Automation (MED). IEEE, 2021.
7) Kodama, Naoki, Kazuteru Miyazaki, and Taku Harada. "A proposal for reducing the number of Trial-and-Error searches for Deep Q-Networks combined with exploitation-oriented learning." 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). IEEE, 2018.
8) Azizzadenesheli, Kamyar, et al. "Surprising negative results for generative adversarial tree search." arXiv preprint arXiv:1806.05780 (2018).
9) V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari With Deep Reinforcement Learning," 2013, arXiv:1312.5602
10) Karol Arndt, Aalto University "Two Player Wimblepong," [online] Accessed from: https://github.com/aalto-intelligent-robotics/wimblepong/tree/
11) Andrej Karpathy, "Deep Reinforcement Learning: Pong from Pixels," [online] Accessed from: http://karpathy.github.io/2016/05/31/rl/