# Reinforcement Learning Project
# WimblePong PPO with CNN

Stefano Baggetto - 879118
Niladri Dutta - 890812
ELEC-E8125 - Reinforcement Learning

February 17, 2021

## 1 Introduction

The project work aims to demonstrate the implementation of a reinforcement learning agent, called *Tumba*, that can play the game of Pong from pixels data(see Figure 1). In this environment, the agent controls one paddle and can take one of three actions: moving up or down, or staying in place.

The aim of Pong is to keep the ball in the game in order to avoid losing. The ball is initialized randomly. It could start moving to the left side or the right side with different angles. The problem must be solved by building a model capable of learning, from the (pixel) states observed, how to act in order to intercept the ball and make it bounce toward the other side. In addition, the agent should also learn how to make the ball difficult to reach for the opponent, so that it increases its chances of winning.

### Objective

The project aims to develop an agent and the relative training process that makes it capable of learning how to play Pong based on pictures observed which represent the current state of the environment. Different models fitting the environment specification must be identified and developed in order to achieve considerable results.

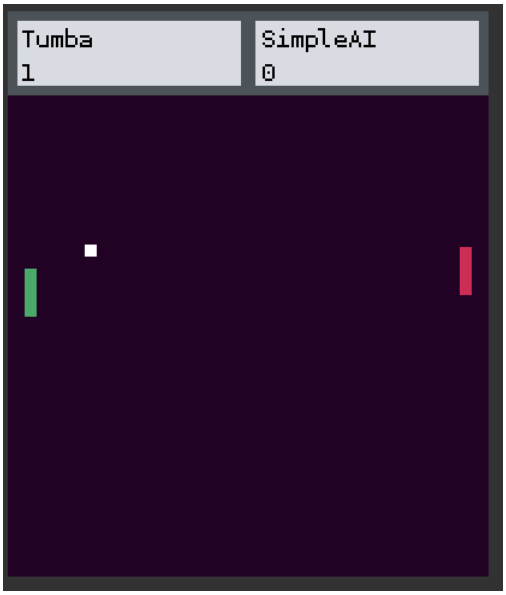


Figure 1: The Pong Environment

## 2 Review of the external sources

During this project, it was allowed to take inspiration from external sources, as long as they are referenced. The model developed in this project refers to external sources, and the reasoning behind the model's design choices are discussed in the later sections.

In order to understand how Proximal Policy Optimization (PPO) Algorithms work, we consulted the paper from Schulman et al. [3]. Another consulted source is the research paper Playing Atari with Deep Reinforcement Learning by Mnih et al [2] contained in the assignment document. The paper describes an approach based on DQN for seven Atari 2600s games from the Arcade Learning Environment. This paper inspired us in the implementation of the architecture of the Convolutional Neural Network(CNN) in the PPO+CNN approach. Furthermore, the repository by Sagar Gubbi pong-ppo [1] allowed a further understanding and better implementation of the loss computation and neural network update in the PPO and, consequently, PPO+CNN solutions.

# 3 Design of the architecture

Following the suggestions in the project document, we approached the problem in different manners, dividing the workload between the teammates and discussing the implementation and the results. In this section we discuss the delivered one.

## Proximal Policy Optimization PPO

One of the implementations attempted was PPO with a fully connected (FC) neural network for the policy. In this method, the loss is calculated by:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[ min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)) \right]$$

Here $r_t(\theta)\hat{A}_t$ is the unclipped objective. The aim is to use the clip(.) term to clip the probability ratio, which penalizes policy changes that move $r_t(\theta)$ outside the interval $[1-\epsilon, 1+\epsilon]$. By taking the minimum of the clipped and unclipped objectives, the final objective becomes the lower bound of the unclipped objective.

Unfortunately, the method performed poorly, the network was not able to learn properly. However, the same strategy including a convolutional neural network (CNN), which is a better approach to deal with images, showed to be an improvement and led to decent results. PPO with CNN is described in the following section.

## Proximal Policy Optimization PPO+CNN

This approach of PPO+CNN is an evolution of the previous PPO+FC method. Despite maybe not being the best solution for Wimblepong environment, it performed decently for the resolution of the problem and it has been selected as deliverable. The following sections aim to give an idea of the reasoning behind the development.

## Preprocessing

The agent plays the game from Pixels, which means that the input consists of frames made of pixels. In particular 200x200 pixels RGB with three channels. The preprocessed images will feed the convolutional neural network. The steps firstly clean the data from irrelevant information received from the environment of the raw game.

- The input 200x200x3 is cropped behind both paddles and only columns of pixels from 9 to 192 are retained Resulting in a 200x184x3 frame.
- Afterwards, the frame is downsampled by a factor of 2 resulting in a 100x92x3 frame which should be easier to interpret without a loss of quality.
- Then, the frame is turned from RGB to grey-scale and the background colour is set to 0.

Finally, in order to allow the network to understand the direction of the ball, the preceding frame is stacked to the current observation. Thus, the network input is two 100x92x3 frames.

## Convolutional neural network architecture

The preprocessed images are in the input to a convolutional neural network. The architecture of the convolutional neural network is inspired by the paper by Mnih et al [2] and is described as follows:

- The first convolutional layer which convolves 16 8x8 filters, stride 4 and activation function ReLU
- The second convolutional layer which convolves 32 4x4 filters, stride 2 and activation function ReLU
- The third is a flatten layer which reshapes the output of the previous layer into a one-dimensional tensor
- The fourth layer is a fully connected layer made of 128 nodes. It connects the input neurons to the next layer and applies ReLU.
- The output layer is again a fully connected layer made of 128 nodes. It connects all the previous output with the final one. The final neurons return the log-odds of every possible action.

Finally, the softmax applied to the log-odds returns the probability distribution of the actions. During the training, the action is sampled from the distribution. During the evaluation, the action with the highest probability is selected.

## Training

The training process involves the storing of the outcomes by the agent, during each timestep. Every 10 games (episodes) they are used to update the policy (parameters). Afterwards, the history of outcome is emptied. During the training, lists and counters (such as episodes, rewards, win rate) useful to create the plots are also populated.

## Update Policy

The updates are made computing a binary mask from the batch states, actions, actions probability and reward. The binary mask represents the one-hot encoding of the action in the batch. Then, the softmax computes the probability distribution of each observation and the mask is applied to it in order to obtain the probability of each action using the current policy. After, the current probability and the policy probability when the action was taken are divided element-wise. So, a list is created by multiplying the ratios by the advantage, and another one is done in the same way, but clamping the ratios between $1 \pm \epsilon$ (where $\epsilon$ is a hyperparameter of the Policy Class). The loss is computed as the mean of the negative element-wise minimum between the two arrays. It is then passed to perform backpropagation on the artificial neural network through the optimizer.

## Hyperparameter Tuning

The hyperparameter tuning was performed through trial and error. The final choice made is based on consultation of the literature, knowledge of the environment, empirical observations and discussion with other teams using similar approaches. Hyperparameter tuning is a tedious, time and resource-consuming task. The time constraints and the number of attempts didn't make it possible to extensively explore and fine-tune the parameters.

| **Hyperparameter** | **Value** |
|---|---|
| Discount($\gamma$) | 0.99 |
| Clipping parameter($\epsilon$) | 0.1 |
| Hidden Neurons | 128 |
| Frame stack size | 2 |
| Learning rate[1] | $10^{-4}, 10^{-5}$ |

Table 1: Hyperparameter Table

## Agent and Policy code

Agent and Policy are the two classes implemented in order to solve the challenge. The class Agent contains the methods to interact with the environment in train and test. The class Policy provides the methods to update the Agent behaviour during the training phase and to get the action to perform in the testing phase.

### The Agent

Agent's constructor defines the parameters and hyperparameters. The agent's initialisation parameter `evaluation` refers to whether the agent is initialized in a training or testing environment. It has been set to `evaluation = True` by default in order to interface with the testing provided. In particular, the effect is to disable the dropout and the batch normalization of the artificial network and to return the action with maximum probability in the `get_action()` method. The device is chosen to be *cpu* but *cuda* would be preferred to exploit the GPU's faster matrix computation. Adam optimizer with a learning rate of $10^{-4}$ was chosen for the first training phase and lowered to $10^{-5}$ in the second phase. Further explanation is provided in the following sections.

- `reset_lists()`: it empties the lists used for computing the loss. The method is called in the training after `episode_finished()`, in the training phase.
- `save_model()` and `load_model()`: Methods to save and load the state of the Policy module. `save_model()` is used in the training phase and `load_model()` to resume it for further training or evaluation.

[1]See Training Methodology section

- `preprocess()`: implementation of the preprocessing discussed in the previous section.
- `get_action()`: Receives the observation (state) as input and, as output, gives a sampled action and its probability (train phase) or the action with maximum probability in case of Evaluation = True (test phase). Firstly, the input is processed as described in the preprocessing section. Then, the preprocessed state is passed to the convolutional neural network and the log odds are computed. Finally, the output is determined according to the evaluation parameter.
- `store_outcome()`: every timestep, it stores the state, the action taken, the probability of the action taken and the reward.
- `episode_finished()`: It updates the weight of the neural network every ten episodes. Firstly, uses the rewards stored in each time step to calculate the discounted rewards. Then the network is updated five times. In order to do that, sampling is performed on the lists stored previously by `store_outcome` and on the advantage. The advantage is calculated as normalized discounted rewards. Afterwards, the loss is computed using the samples by the `forward()` method in the Policy class. Finally, backpropagation is performed using the loss through the optimizer.

### The Policy

Policy is an attribute of Agent but it worths a separate description. As for the class Agent, the constructor defines the parameters and hyperparameters. The device is set to accord to the Agent's one. `ep_clip` is used from the clamping in the loss computation. Hidden neurons, `reshape_size`, convolutional layers and fully connected layers define the architecture of the neural network together with the convolutional layers. Following, the Policy class methods are presented.

- `init_weights()`: it load the weights saved by `Agent.save_model()` and collected by `Agent.load_model()`
- `layers()`: it forward propagates the input in the network.
- `forward()`: it receives the batches form `Agent.episode_finished()` and computes the loss function.

# 4 Training Methodology

The training process needed for the agent to learn how to play the game and win against the opponent has been carried to maximize the win rate against the opponent, represented by the SimpleAI. It consisted in two phases for a total of 108 000 games (episodes) played. The phase difference resides in the optimization methods, chosen based on consultation of the literature, knowledge of the environment and empirical observations, but also in trial and error manner.

## Phase 1

In the first phase Adam optimizer has been adopted and the learning rate has been set to $10^{-4}$. Due to an error in the machine, the train stopped at 94,000 episodes. The figure shows how the rewards seem to stop growing after (indicatively) 60,000 episodes but the winning rate seems to slowly but constantly growing. For this reason, the model produced after 94,000 iterations have been used for further learning.

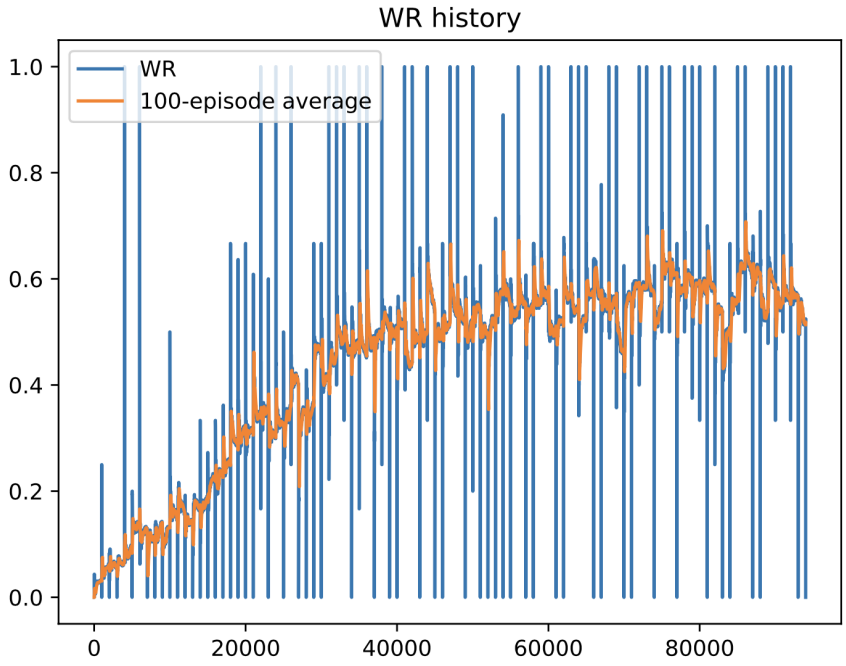


Figure 2: The training phase 1

## Phase 2

In the second phase, the model produced after 94,000 iterations have been used for further learning. Specifically, three approaches have been adopted:

- Adam optimizer with learning rate $10^{-3}$. The intuition behind raising the learning rate is trying to let the optimization escape from a local minimum. Unfortunately, this process resulted in no model improvements and has been stopped.
- Adam optimizer with learning rate $10^{-5}$. On the contrary of the previous intuition, with a smaller learning rate, we wanted to explore deeper. The outcome of this approach was a gradual slow growth of the rewards and the win rate (Figure). The process stopped after 14,000 more episodes, and the model resulted from this training is the one delivered.
- SGD with learning rate $10^{-4}$. A change of the optimizer aimed to discover if different approaches could improve the results. This step showed no improvements.

Summarizing the training consisted of 94,000 training episodes with Adam optimizer and learning rate $10^{-4}$, followed by 14,000 training episodes with Adam optimizer and learning rate $10^{-5}$.

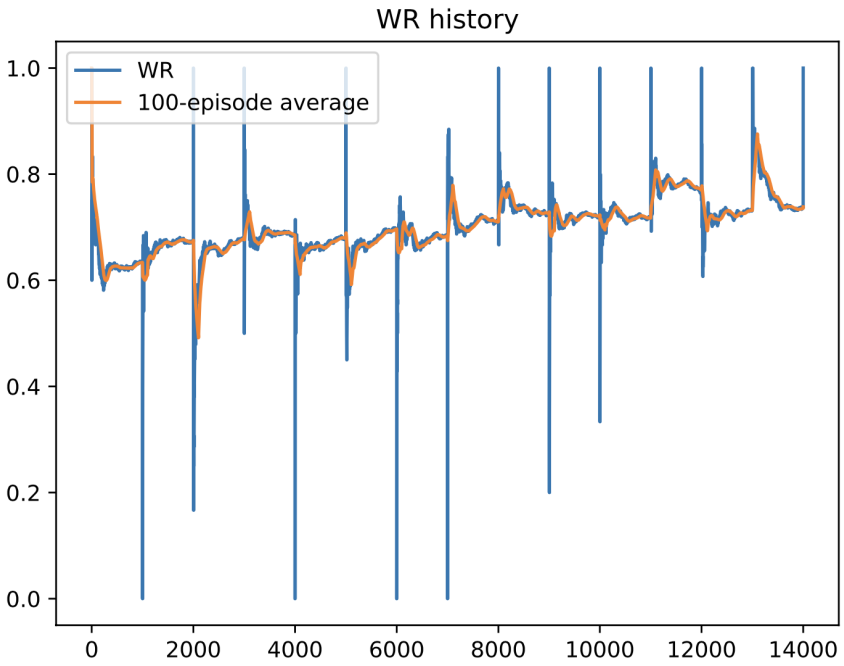


Figure 3: The training phase 2

# 5 Evaluation of results

The figure shows some testing of the trained agent against SimpleAI, the average win rate in the tests carried out measures: 73.4%.

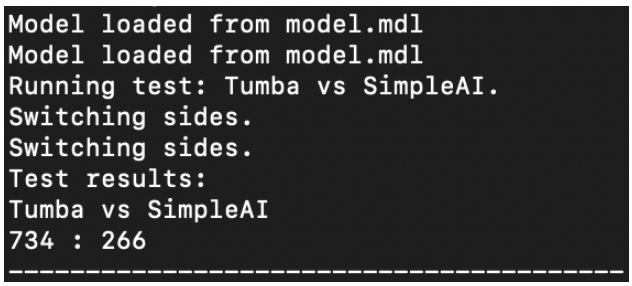


Figure 4: Test: Tumba vs Simple AI

Besides, The visualization of the matches of the agent against SimpleAI allowed us to notice some of the agent's behaviour. Firstly the agent seems to predict where the direction of the ball and how to intercept it. However, if the ball has a close to vertical angle, the agent does not move as fast to reach it. The agent sometimes follows the position of the ball after hitting it, emulating the behaviour of SimpleAI. To score a point the agent hits the ball with an edge of the paddle, making the angle more vertical and more difficult for the opponent to reach it.

## Improvements

The results obtained are satisfactory against SimpleAi. However, other improvements have been considered and might be further implemented and tested. More frames could be fed into the preprocessing to let the agent better recognize more complex patterns of the ball and the opponents. The training process could be sped up through the usage of frameskip as suggested in the DeepMind paper. [2] To exclude overfitting against an opponent, the agent could be trained against other agents, trained with different approaches or even against itself.

# 6 Conclusions

The general performance is acceptable against SimpleAI even though, as explained in the last section, improvements can be made. The PPO approach with a convolutional neural network might not be the best approach for developing an agent playing Pong. However, it is the one which showed growing performance and therefore also the one which has been further researched and tuned. Due to time constraint, the other suggested methods couldn't receive the same amount of resources. Furthermore, even performing development and training in parallel in multiple machines, the training process was always a bottleneck as the number of machines available and equipped to process the training is limited, and sudden stops can occur. To ease this problem, some credit for online computing services could be provided in the future.

# References


[1] Sagar Gubbi. *pong-ppo.* `https://gist.github.com/s-gv/b13974f896c7baf81ea3a83cf1af4a66`.

[2] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: `1312.5602 [cs.LG]`.

[3] John Schulman et al. *Proximal Policy Optimization Algorithms.* 2017. arXiv: `1707.06347 [cs.LG]`.