ELEC-E8125 - Reinforcement Learning
Aalto University

COURSE PROJECT
PONG

Chiappetta Antonio 801720

Molteni Luca 801775

December 9, 2019

Contents

1 Introduction

1.1 Problem description: Pong from pixels

Pong (1) is a table tennis sports video game using simple two-dimensional graphics. The game was initially released by Atari in 1972. In Pong, the player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The goal is for each player to reach eleven points before the opponent; points are earned when one fails to return the ball to the other.

The game of Pong is an excellent example of a simple RL task (2). In the Wimblepong environment developed for the course project, based on the one provided in OpenAI Gym (3), we used our agent to control one of the paddles (the other is moved by a decent AI) with the objective of bouncing the ball past the other player. On the low level the game works as follows: we receive an image frame (a 200x200x3 byte array (integers from 0 to 255 giving the RGB values of the image pixels)) and we get to decide whether to move the paddle UP, DOWN, or keeping it in place. After every single choice, the game simulator executes the action and gives us a reward: Either a +10 reward if the ball went past the opponent, a -10 reward if we missed the ball, or 0 otherwise. And of course, our goal is to move the paddle so that we get high rewards.

1.2 Project task

In this course project, our objective was to use a reinforcement learning algorithm to teach an agent how to play Pong from raw images. The task included the application of the concepts learned during the course and the exploration of state-of-the-art techniques in the field.

We could reuse external sources (including code) and adapt them to the needs of our environment Wimblepong, designed after OpenAI Gym's Pong environment, in order to provide us with an easy to use version of the game.

We were also allowed to preprocess the raw images, e.g., by color transformations, stacking multiple frames together, or by a supervised learning model, but extracting the positions of elements (e.g., by color) was not allowed.

2 Approach and method

We started by following two main paths:

- Start from the main resource provided in the instructions for the project (2) and try to adapt to our environment the relative REINFORCE algorithm in (4) and its improvement with an Actor Critic approach in (5)
- Use two of the most advanced techniques studied during the course, DQN and Actor Critic, and adapt the code produced by ourselves during the weekly assignments to this case. For DQN, the theoretical reference was provided in (6), while for the Actor Critic we referred again to (2)

The adaptation of the REINFORCE algorithm proposed in (4) didn't prove to be effective, having scored no more than 8% against SimpleAi after a training process of 60000 episodes. For this reason,

we decided to move to the Actor Critic solution proposed in (5) as an improvement of the previous one. But also in this case the solution performed in a similar way.

Having studied more intensely the drawbacks of Policy Gradient algorithms in terms of how bad moves can ruin the performance of training (7), we decided to try implementations of TRPO (Trust Region Policy Optimization) (8), and towards the end of the project, of PPO (Proximal Policy Optimization). In both cases, the adaptation proved to learn very slowly and we decided to move on. We also tried to adapt an implementation of ACER (Actor Critic with Experience Replay) provided in (9), but the results were similar to the previous two. In all these cases, we believe that performance should have been improving from algorithm to algorithm, but our lack of practical experience and the differences between our environment and the one offered by Gym (3) have prevented us to come up with the right learning process. Most of the sources we referred to propose very detailed and efficient implementations, but it has also been complicated to understand what to modify in order to make them work in our case.

What proved to perform more effective was the implementation of the algorithms learnt during the assignments. For DQN, results started improving, but the algorithm was taking a lot of time to learn and never went above a. 22% victory rate.

The implementation of Actor Critic based on the 5th assignment of the course was the one performing better, after being trained using as input the state of the game, modeled by the positions of the ball and the two paddles. At that point, we realized we could use supervised learning and try to predict these data from the raw image made of pixel data. The implementation of this step with a Neural Network proved to be successful, and predictions quickly reached satisfying results with an average error of 1-2- pixels, thus only slightly disturbing the behaviour of the Actor Critic algorithm.

After having trained the agent against SimpleAi, this implementation has guaranteed a performance that, from game to game, ranged between 35% to 55% victory rate against SimpleAi. This was the final agent and model submitted for the evaluation. We also tried to train the agent against himself or against the agent named SomeAgent provided by the course instructors, but in both cases, our agent did not learn a behavior that could be effective against SimpleAi.

The last approach that we tried to implement was an implementation of the Experience Replay buffer in our Actor Critic agent, using a sample of recently played episodes (and not just one) for each policy gradient update. However, in the little time remained, it did not learn a useful policy and we did not have time to tune it perfectly.

3 Image pre-processing

Since the context of our task required to build a Reinforcement Learning model explicitly taking images as input, being able to pre-process the raw image in a good way is a key step towards a winning agent.

3.1 Dimensionality reduction

As previously mentioned, the raw observation returned by the environment, while in visual mode, is a (200 * 200 * 3) values multi-dimensional array of integers between 0 and 255.

The original Open-AI *Pong-v0* environment (3) actually returns a 210 * 160 * 3 array because it also includes the portion of the screen showing the scores. To our convenience, the custom environment used for the project is already dealing with this step and the original frame of size 230 * 200 pixels is already cropped.

Since the Agent takes the whole observation as input, and given its high dimensionality (120000 values), it sounds reasonable to try to reduce the size of the observed frame.

The Pong game has a clean and simple graphical interface. There are only two rectangular-shaped paddles with size 5 * 20 pixels and one squared ball of 5 * 5 pixels. Given these considerations, we usually scaled the image by a factor of 2, without incurring the risk of losing the information contained in the screen snapshot.

3.2 Decolorizing

There are only 4 different colors in Pong: for the ball, for the left paddle, for the right paddle and the background.

Since the main elements of interest are the ball and the paddles' position we tried to magnify the salience of these elements by de-colorizing the image. The G (green) and B (blue) layers got discarded and the R (red) value of the background has been set to 0, while all other values, ball, and paddles, have been set to 1.

3.3 Frame stacking

Finally, in some of the approaches, we applied frame stacking. One static snapshot of the game screen by itself does not provide any valuable information about the dynamics of the game. The observations can be enriched by stacking together multiple previous frames to intrinsically add information about the trajectory and the speed of the ball.

We tried either to stack 2 or 4 frames together or to subtract the values of the previous observation from the current one as seen in (4).

4 Algorithms

In the following subsections, we are going to describe the algorithms we have/have tried to implement with particular emphasis on the adjustments applied to make them work with the given environment.

4.1 DQN - Deep Q-Learning

The DQN algorithm is a famous approach proposed in 2013 to try to effectively apply the use of deep neural networks in Reinforcement Learning tasks. The original paper (6) implemented and tested a model to play to several games part of The Atari Learning Environment (10), including Pong; thus, we considered this approach right at the beginning of our project. In addition, we

already experimented with the DQN algorithm during the course. More specifically, assignment 4 tasked to solve the cart-pole and lunar lander environments using this approach. For this reason we developed our solution starting from the implementation partially given in the assignment.

The DQN algorithm can be seen as a variation of the Q-learning algorithm (reference) where we try to train a model that approximates the Q-values of all action given a state (6). This approach is commonly used in task with a continuous space state and in the classic Q-learning approach the Q-values are computed by some sort of linear function approximator. In the case of DQN, the approximator is a neural network, called Q-network, that uses stochastic gradient descent to update the weights. Other characteristics of DQN is the use of an experience replay mechanism, which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors reducing the correlation between the data with benefits both for the stability of the algorithms and the data efficiency. It is a model-free algorithm, meaning that no explicit representation of the dynamics of the environment is modelled and it is also off-policy. During training a sufficient amount of exploration in granted using an ϵ-greedy strategy. The greedy action is selected with probability 1 ϵ and a random action with probability ϵ. The greedy strategy is GLIE (Greedy in the Limit with Infinite Exploration), meaning that with time approximating to infinite the probability of exploring decreases to 0 granting convergence to a deterministic policy.

The main difference between the code that was provided in assignment 4 and the approach followed by us and by the original paper, is that the network is trained not on some specific state values but on the raw video data coming from the game. As a consequence, we added some convolutional layers to the network and a function to pre-process the input image. AN additional data structure, named History as been added to keep track of the last n frames of the game in order to stack images together and obtain partial game sequences.

```
class History(object):
    def __init__(self, size):
        self.size = size
        self.frames = []

    def get(self):
        return self.frames

    def put(self, frame):
        if len(self.frames) == self.size:
            self.frames.pop(0)
            self.frames.append(frame)
        else:
            self.frames = [frame] * self.size

    def empty(self):
        self.frames = []
```

Listing 1: History object

4.1.1 Preprocessing and Model Architecture

The input to the neural network consists is an 80 80 4 image produced by the *preprocess*() function. The first hidden layer convolves 16 8 8 filters with stride 4 with the input image and

applies a rectifier non-linearity. The second hidden layer convolves 32 4 4 filters with stride 2, again followed by a rectifier non-linearity exactly as specified in the original paper. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each of the 3 actions (MOVE UP, MOVE-DOWN, STAY).

The outputs correspond to the predicted Q-values of the individual action for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.(6)

As explained in the introductory section the state observation returned by the environment is a 200*200*3 multi-dimensional array of int values. In order to conform to the input defined for the NN we first take form the History the last n frames. Then we convert the image to grey-scale and resize it to a size of 80 * 80 pixels. Here follow the code snippet of the *preprocess*() function.

```
    def preprocess(self, observation):

        # ADD obs to history
        self.history.put(observation)

        if self.prev_obs is None:
            self.prev_obs = observation

        img_list = self.history.get()

        self.prev_obs = observation

        k = len(image_list)

        im_tuple = tuple()

        for i in range(k):

            # Load single image as PIL and convert to Luminance
            pil_im = PIL.Image.fromarray(image_list[i]).convert('L')
            # Resize image
            pil_im = pil_im.resize((80, 80), PIL.Image.ANTIALIAS)
            # Transform to numpy array
            im = np.array(pil_im) / 255.
            pil_im.close()
            # Add processed image to tuple
            im_tuple += (im,)

        # Stack tuple of 2D images as 3D np array
        arr = np.dstack(im_tuple)
        # Move depth axis to first index
        arr = np.moveaxis(arr, -1, 0)
        # Make arr 4D by adding dimension at first index
        arr = np.expand_dims(arr, 0)
        return arr
```

Listing 2: preprecessing function

4.1.2 Parameters, training and results

Here follows the parameters used with the most promising model:

- TARGET_UPDATE = 20
- glie_a = 5000
- gamma = 0.95
- replay_buffer_size = 100000
- history_size = 4
- batch_size = 256
- learning_rate = 10^{-3}

We initially tried with a smaller replay memory of 50000 samples but we decide to increase it to further reduce the correlation between data at the cost of increased computational requirements. The initial batch size of 128 has also been increased to 256 after enlarging the replay memory. The first model used only 2 stacked frame while the second one had them increased to 4 as indicated in (reference). We trained both models for 5000 episodes with similar results. Both of them reached a victory rate against SimpleAI of around 20%. On the one side we consider it a success since the model was clearly learning to play the game, on the other side we haven't been able to further increase the performance neither with additional training episodes or tweaking the learning rate.

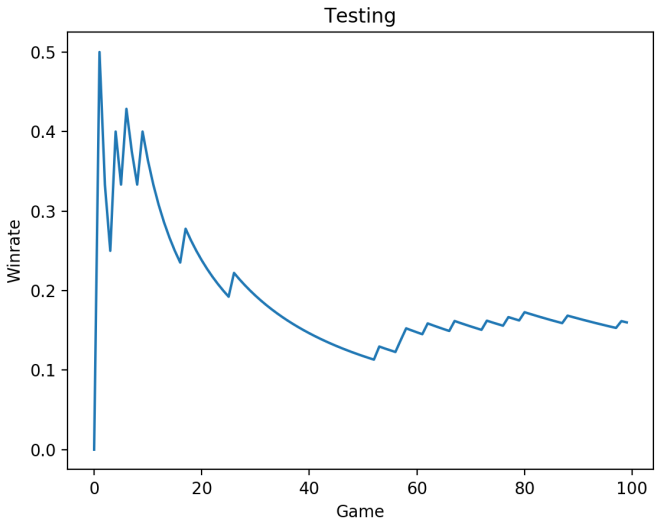

Figure 1: Average winrate after 100 games of testing

This model was extremely slow to train thus we decided to abandon the DQN approach to explore alternative solutions.

4.2 REINFORCE

REINFORCE belongs to the family of policy gradient algorithms.

The goal of reinforcement learning is to find an optimal behavior strategy for the agent to obtain optimal rewards. The policy gradient methods (11) target at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function with respect to θ, $\pi_\theta(a|s)$. The value of the reward (objective) function depends on this policy and then various algorithms can be applied to optimize θ for the best reward. The reward function is defined as:

$$J(\theta) = \sum_{s \in S} d^\pi(s) V^\pi(s) = \sum_{s \in S} d^\pi(s) \sum_{a \in A} \pi_\theta(a|s) Q^\pi(s,a) \tag{1}$$

where d^π is the on-policy state distribution under π.

Using gradient ascent, we can move θ towards the direction suggested by the gradient $\Delta_\theta J(\theta)$ to find the best θ for π_θ that produces the highest return.

The computation of the gradient is tricky because it depends on both the action selection (directly determined by π_θ) and the stationary distribution of states following the target selection behavior (indirectly determined by π_θ). Fortunately, the policy gradient theorem simplifies the computation by removing the derivative of the state distribution.

REINFORCE (Monte-Carlo policy gradient) relies on an estimated return by Monte-Carlo methods using episode samples to update the policy parameter θ. REINFORCE works because the expectation of the sample gradient is equal to the actual gradient:

$$\begin{aligned} \Delta_\theta J(\theta) &= E_\pi[Q^\pi(s,a) \Delta_\theta \ln \pi_\theta(a,s)] \\ &= E_\pi[G_t(s,a) \Delta_\theta \ln \pi_\theta(A_t, S_t)] \end{aligned} \tag{2}$$

Therefore we are able to measure G_t from real sample trajectories and use that to update our policy gradient. It relies on a full trajectory and that's why it is a Monte-Carlo method.

The process is pretty straightforward:

- Initialize the policy parameter θ at random
- Generate one trajectory on policy
- For each step of the trajectory, estimate the return and update policy parameters $\theta = \theta + \alpha\gamma^t G_t ln\pi_\theta(A_t, S_t)$

We tried to implement the version of REINFORCE proposed in (4) but it never learnt a behaviour granting a victory rate over 9%.

4.3 Actor Critic

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients, and that is exactly what the Actor-Critic method does.

Actor-critic methods consist of two models, which may optionally share parameters:

- Critic updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a, s)$ or state-value $V_w(s)$.
- Actor updated the policy parameters θ for $\pi_\theta(a, s)$, in the direction suggested by the critic.

After having tried the implementation of REINFORCE in (4), we tried also the version of actor critic that was built on top of that in (5), but it got similar, not satisfying results.

The version realized instead following the work we did in the assignment proved to be the most effective one and became part of the final solution.

4.4 TRPO - Trust Region Policy Optimization

The policy gradient methods explored before face a few challenges that hurt their performance.

First, PG computes the steepest ascent direction for the rewards (policy gradient) and update the policy towards that direction. However, the usage of the first-order derivative approximates the surface to be flat. If the surface has high curvature, we can make horrible moves. On the other hand, if the step is too small, the model learns too slow.

Second, it is very difficult to set a proper learning rate and avoid suffering from converging problem.

And third, doing one update for each trajectory is not sample efficient.

As shown in (12), the way that TRPO uses to solve these problems is to constraint the policy changes to avoid too aggressive moves. A trust region is defined as the circular region around the current size whose radius is determined by the maximum step size that we want to explore.

The algorithm guarantees monotonic improvements, creating a better policy at each iteration. Intuitively, as explained in (7), we are approximating the expected advantage function locally around the current policy with an accuracy that decreases when the new policy and the current one diverge from each other. But, since we can establish an upper bound for the error, we can guarantee a policy improvement as long as we optimize the local approximation within a trusted region.

We tried to implement the solution in (8), but is showed a very slow learning.

4.5 ACER - Actor Critic with Experience Replay

As shown in (13), the usage of experience replay in off-policy learning can be the right strategy to improve the sample efficiency of actor critics. Variance and stability of off-policy estimators are however still hard to control.

The approach proposed in (13) and the implementation in the suite of OpenAI baselines provided at (9) has been a source of inspiration for us to implement the experience replay technique in our A2C, performing as our best model.

We tried to use a different way of updating our policy, replacing the single update per trajectory with an update comprising a batch of recent trajectory sampled by a large experience buffer. The algorithm showed slow learning capabilities and considered the very short time remaining we did not tune it more to finally stick on the A2C approach based on single trajectory updates.

4.6 PPO - Proximal Policy Optimization

PPO seek a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation

from the previous policy is relatively small. (14)

PPO is an algorithm introduced by OpenAI which, as other approaches such as TRPO and ACER, tries to constrain the size of a policy update. (15) The main advantages is represented by a much more simple implementation. ACER, for example, requires the addition of code for off-policy corrections and a replay buffer. TRPO is not easily compatible with algorithms that share parameters between a policy and value function, like the one we used in our visual Pong problem. A good probability of optimizing within a trust region is granted by adding a soft constraint to the objective function.

The version of PPO we tried to implement is the so-called PPO with Clipped Objective

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

where

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ϵ is a hyperparameter, usually 0.1 or 0.2

We focused our attention on this algorithm only during the last day of the project. We were not able to improve further our current solution, thus we decided to explore a new solution that apparently showed excellent results for some groups. We implemented the algorithm taking inspiration from this (16) blog post. After running the model for around 5000 epochs we noticed that it was slowly showing some improvements. Our best result before deciding that there was no time to optimize/tune the algorithm, was only around 10%.

5 Final Solution and performance analysis

I the following section we are going to describe the algorithms that we decided to submit since it gave us the most promising results in the testing phase. It can be described as an A2C model whose inputs are fed by the results extracted by a number of Neural Networks who themselves took the image frames as input. We decide to try this algorithm mainly for two reasons. First, during the project, we were playing around a bit also with the non-visual version of the environment and noticed that we were able to reach interesting results with an A2C approach. Secondly, we tried to use NN on the image frames and noticed that we were able to extract with extreme precision some game-specific information such as the position of the ball. The next subsections are going to be organized as follows: one section will be dedicated to describing how we structured and trained the NN used for information extraction, the next section will briefly describe the parameters on the training of the subsequent A2C algorithm. Finally one section will present the overall results of the model.

5.1 Information extraction with NN

As previously mentioned we used some NN to extract some game-specific features from the images of the game. The data we were able to retrieve with a satisfactory degree of precision were the following:

- Y-coordinate of the ball
- Y-coordinate of the player's paddle
- Y-coordinate of the opponent's paddle

We were not able to extract with a sufficient precision margin the X-position of the bull thus the subsequent A2C model will not use this information.

5.1.1 Network structure

The first design decision the was taken regarding the network architecture was to train for each feature a different network with just one output. The input of the model was a 100000 (100x100) 1D float vector obtained from the 200*200*3 images returned by the environment. The G and B layers were removed from the original images, it was then resized by a factor of 2 and finally, the background was set to 0 while the rest (ball, paddles) to 1. There was no need to add any convolutional layer because we do not want to enhance any characteristic of the images but rather preserving the exact position of a precisely shaped object.

We tried 2 different architectures to perform the task. The first one, which proved to be the most effective, was formed by two fully connected hidden layers. There were 10000 input features, 256 neurons in the first hidden layer, followed by a Rectifier Linear Unit, and 64 neurons in the second fully connected hidden layer, again followed by ReLU. Finally, the output layer only had 1 output corresponding to the trained feature.

The alternative architecture that was tested included only one hidden layer with 100 neurons followed by ReLU.

5.1.2 Training, Validation and Testing

We applied a supervised learning approach to solve the task of extracting some game-specific information because we were able to generate a training data-set using the environment attributes and functions.

The training validation and testing sets were generated as follows. Two simpleAI agents were let playing against each other indeterminately. At each step, the observation captured by the first player was stored as a sample with a probability p. The infinite playing loop gets interrupted when enough samples are gathered. The parameters used for data generation are the following:

- train_set_size = 20000
- val_set_size = 5000
- test_set_size = 1000
- sampling_prob = 0.1

We think this was a good approach to generate the dataset for the supervised algorithms because it tried to reduce the presence of highly correlated data (by sampling with low probability) and by letting 2 SimpleAi agents playing against each other. All the obtained sets have been then shuffled. The training and validation datasets have been inserted into a custom Dataset object used later for batch sampling.

```
from torch.utils.data import Dataset

class PongDataset(Dataset):
    """Pong images dataset."""

    def __init__(self, numpydata):
        self.pong_frames = numpydata

    def __len__(self):
        return len(self.pong_frames)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        y = self.pong_frames[idx][1]
        frame = self.pong_frames[idx][0]
        sample = (frame, y)

        return sample
```

Listing 3: PongDataset object

Training and Validation Datasets were then given to a SubsetRandomSampler which was subsequently given as a parameter to a DataLoader able to extract sample batches of specified size.

- train_batch_size = 32
- val_batch_size = 128

Here follow the results of the training phase with:

- leaning_rate = 0.001
- n_epochs = 250

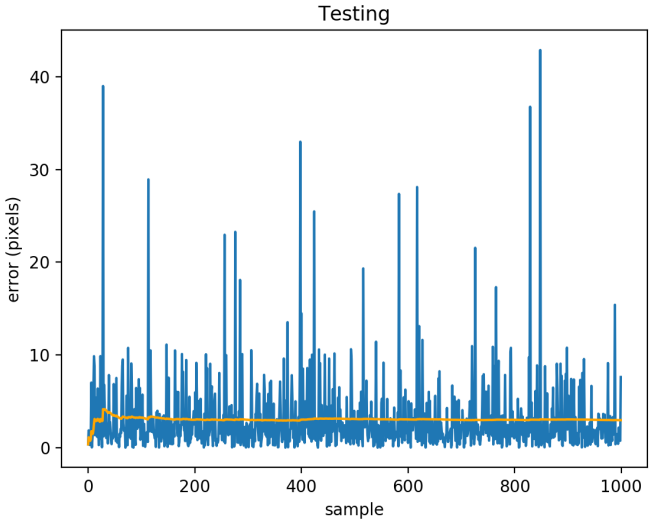

Figure 2: Testing error (blue) and avg error (orange) on 1000 samples for the Y-coordinate of the ball

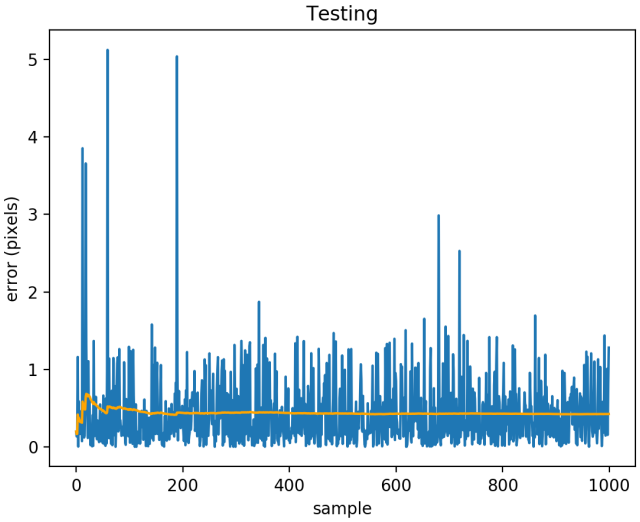

Figure 3: Testing error (blue) and avg error (orange) on 1000 samples for the Y-coordinate of the player's paddle

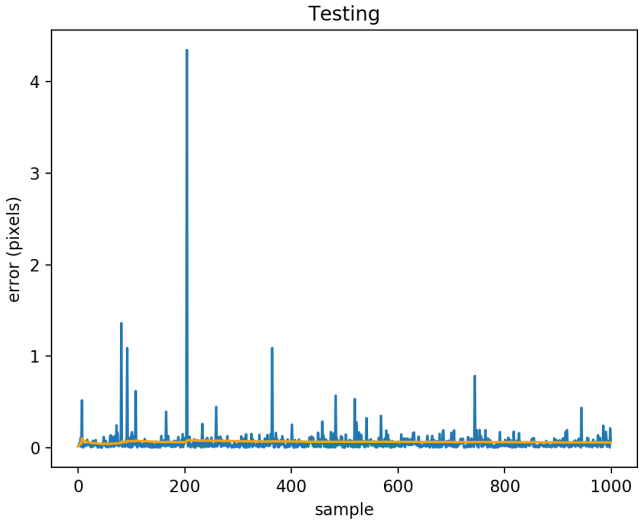

Figure 4: Testing error (blue) and avg error (orange) on 1000 samples for the Y-coordinate of the opponent's paddle

5.2 A2C implementation

Our A2C agent uses the output predictions of the previous preprocessing layers to take decisions on whether to move up or down.

5.2.1 Policy network

The agent's policy network is used to update the policies of both the actor and the critic component. In both cases, the input layer has the size of the state space, where the state is identified by the current vertical positions of ball and paddles, and the previous vertical position of the ball (to keep track of motion). Information on the horizontal movement of the ball is not used since it was difficult to predict it with the NN, but also because the performance of this component did not change drastically when removing this information from the training process.

Both sub-networks have their input layer followed by a Rectifier Linear Unit and a hidden layer composed by 64 units. The actor output is a categorical distribution generated by a softmax activation function with probabilities for each action in the action space. The critic output is a single number estimating the value of the state observed.

5.2.2 Training process

The training process was performed in the non-visual version of the Wimblepong environment, where the system could receive as an input the exact coordinates of the game elements and use the RMS optimizer to calculate the gradient and optimize the policy network.

Actions are taking by sampling from the categorical distribution returned by the policy network for the observed state. At the end of each episode, all observed action probabilities, rewards, and state values are used to compute the advantages and, consequently, the actor loss and critic loss. These two had the same weight in computing the total loss. Rewards were discounted and normalized.

The agent was trained for 80000 episodes with the following hyper-parameters:

- Discount factor $\gamma = 0.98$
- Learning rate $\alpha = 5e - 3$

The following plot shows the performance of the training process in the last 20000 episodes.

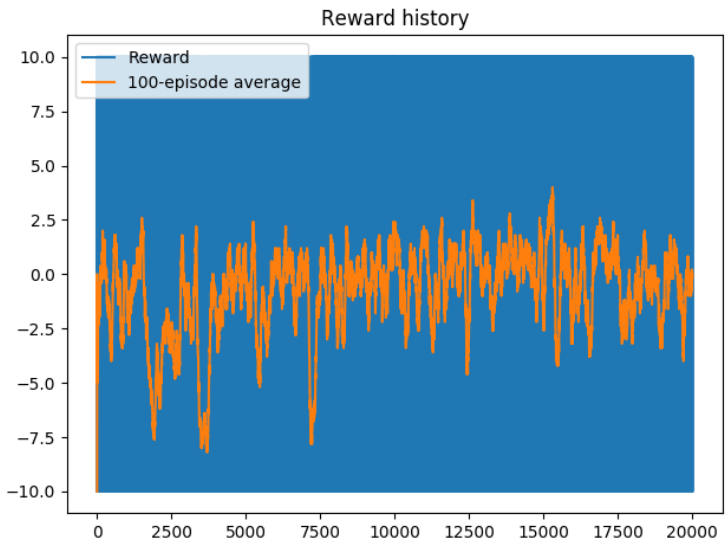

Figure 5: Training process for A2C, last 20000 in a total of 80000 episodes

5.2.3 Usage

The agent receives a raw image and uses the NN structure to predict the vertical position of the ball, player's paddle and opponent paddle. It then forwards this information to the policy network to receive a categorical distribution of action and their probabilities and decides to take the action maximizing the log probability.

5.3 System performance

We ran multiple executions of the test_agent.py script to test our agent against SimpleAi. In its final version, that we delivered as project outcome, it played 20 trials of 100 matches, obtaining a victory rate ranging from 35% to 55%.

6 Repository

All the code used during the development of this project can be found on GitHub at (17).

References

[1] "Pong," https://en.wikipedia.org/wiki/Pong.

[2] A. Karphathy, "Deep Reinforcement Learning: Pong from Pixels," http://karpathy.github.io/2016/05/31/rl/, 2016.

[3] OpenAI, "Pong-v0," https://gym.openai.com/envs/Pong-v0/.

[4] A. Karpathy, "Training a Neural Network ATARI Pong agent with Policy Gradients from raw pixels," https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5, 2016.

[5] W. Xuguang, "Trains an agent with (stochastic) Policy Gradients(actor-critic) on Pong. Uses OpenAI Gym," https://github.com/schinger/pong_actor-critic, 2016.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," pp. 1–9, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[7] J. Hui, "RL — Trust Region Policy Optimization (TRPO) Explained," https://medium.com/@jonathan_hui/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9, 2018.

[8] M. Acar, "PyTorch implementation of TRPO," https://github.com/mjacar/pytorch-trpo, 2017.

[9] M. Agarwal, "OpenAI Baselines," https://github.com/meagmohit/RL-playground/tree/master/baselines, 2018.

[10] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, p. 253–279, Jun 2013. [Online]. Available: http://dx.doi.org/10.1613/jair.3912

[11] R. S. Sutton and G. Barto, *Reinforcement Learning.* The MIT Press, 2015. [Online]. Available: http://incompleteideas.net/book/RLbook2018.pdf

[12] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1889–1897, 2015.

[13] Z. Wang, V. Mnih, V. Bapst, R. Munos, N. Heess, K. Kavukcuoglu, and N. De Freitas, "Sample efficient actor-critic with experience replay," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, no. 2016, 2019.

[14] J. Schulman, "Proximal policy optimization," Mar 2019. [Online]. Available: https://openai.com/blog/openai-baselines-ppo/

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," pp. 1–12, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[16] [Online]. Available: http://www.sagargv.com/blog/pong-ppo/

[17] L. Molteni and A. Chiappetta, "PongRL-ChiappettaMolteni," https://github.com/Molteh/PongRL-ChiappettaMolteni.