Final Project Report

Giulio Marcon - 914756
Mattia Molon - 880615
ELEC-E8125 - Reinforcement Learning

December 6, 2020

Contents

1 Introduction

This report describes the implementation of a Reinforcement Learning agent for the game Pong. Pong is a two-dimensional game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. Players use the paddles to hit a ball back and forth. The goal is score points by making the opponent fail to return the ball. The scope of our agent is to learn how to control one of the paddles in order to win games against simpleAI. SimpleAI is a secondary agent that plays the game by simply following the ball on the y axes of the screen. Even though SimpleAI is able to access the environment variables such as the coordinates of the ball on the play field, our agent has not this capability. On the contrary, it must be able to learn how to play by simply looking at the frames returned by the environment.

In this report, we describe our approach to the problem in Section 2, we report the performance of our agent in Section 3, we describe possible improvements in Section 4, and we finally discuss our work in Section 5.

2 Approach and method

We face the described task with a DQN approach with experience replay, by using a variant of the DQN algorithm implemented in [1]. Our algorithm differs from the original one in the loss function, the optimizer of the neural network, and the utilization of a target network to stabilize the training. All the implementation details will be discussed in Section 2.2. We have opted for this solution mainly because of the high performances claimed in the paper, in which expert human player performances are reached.

2.1 Environment details

The interaction between the environment ϵ and the agent is done via a sequence of actions, observation, and rewards. At each time-step, the agent chooses an action from a pool of legal actions $A = \{UP, DOWN, STAY\}$ and passes it to ϵ. At this point, the environment evaluates the action, updates its internal state, and returns both a numeric reward and a new game frame representing the new state reached. The returned frames consist of a 200x200x3 RGB images that only include the play field, the score board is automatically removed by the environment. The returned rewards are 0 when the ball is in the middle of the frame, 10 when the agent scores a point, and -10 when the opponent scores a point. Therefore, the agent will obtain useful information for its weights update only when the games end in favor of one of the two players.

2.2 DQN with experience replay

We decided to treat the problem as a Q-learning problem. Therefore, for each state s we observe from the game, we try to estimate the $Q(s, a)$ of all the legal a available in ϵ. The policy is then derived by the actions with the maximum Q-value in each state. In order to estimate the $Q(s, a)$ we use a single convolutional neural network with an output layer of dimension 3, one for each possible action (the details of the architecture are presented in Section 2.5). Therefore, the neural network tries to estimate all the Q-values in a state simultaneously. We then use value iteration to update the values derived from the network. More precisely, the Bellman equation is defined as:

$$Q^*(s, a) = E_{s' \sim \epsilon}[r + \gamma \max_{a'} Q^*(s', a')|s, a]$$

Where Q^* represents the optimal Q-value, s' and a' are the next state and next action, r is the rewards we have obtained from taking action a in state s and finally, γ defines the discount factor. Since we do not have access to the real Q-values, and we approximate them via a neural network with parameter θ, the Bellman equation becomes:

$$Q(s, a|\theta) = E_{s' \sim \epsilon}[r + \gamma \max_{a'} Q(s', a'|\theta)|s, a]$$

In order to estimate the optimal Q-value function we use value-iteration. Value-iteration tries to estimate the optimal Q-value function with an iterative process by defining a Q-value at time t as a function of the Q-value at time $t - 1$. More precisely we compute:

$$Q_t(s, a|\theta_t) = E_{s' \sim \epsilon}[r + \gamma \max_{a'} Q_{t-1}(s', a'|\theta_{t-1})|s, a]$$

Therefore, to train the model we minimize the difference between a current prediction and a prediction made with old parameters. Therefore, our loss function is defined as:

$$L_t(\theta) = |(r + \gamma \max_{a'} Q_{t-1}(s', a'|\theta_{t-1}) - \max_{a} Q_t(s, a|\theta_t)|$$

While training the model, we make use of an experience replay memory. We store every new transition $T_t = (s_t, a_t, r_t, s_{t+1})$ we observe from the environment in a memory $M = T_1, ..., T_n$. Then, during training, we sample random batches of transitions and train the model on these batches. The experience replay memory helps the agent to train multiple times on old observations, and helps breaking the dependency between consequent transitions.

To help the training being more stable, we use a target net. Hence, to compute the $Q_{t-1}(s', a'|\theta_{t-1})$ in the Bellman equation, we do not use the network we are using to choose the actions (that we will call policy_net), but we use a copy of this network called target_net. Despite the policy_net is updated at every new transition observed, the target_net is updated every C episodes. This trick should decrease the similarity between the networks that estimate the Q-values in the Bellman equation, create more meaningful updates, and reduce the variance in the training.

As a final remark, to help the agent explore more, we use an ϵ-greedy policy with GLIE. Moreover, a preprocessing procedure Φ and a memory buffer are used to transform the observed frames into more informative stack of images for the agent. The entire preprocess pipeline is described in Section 2.3 and Section 2.4. A pseudo-code describing the full algorithm used is shown in Algorithm 1:

2.3 Pre-processing of a game frame

The observations returned by ϵ consists of 200x200x3 RGB images. In order to help the DQN algorithm to converge faster and learn a significant representation of the observations, we pass them through a preprocessing fucntion Φ before feeding them to the network or save them into the memory. The steps that Φ follows are:

- Downscale the image to 100x100x3 pixels. This step reduces the input dimensionality and thus the computation complexity of the neural network. As a consequence, the training results much faster and converges quicker.

- Black and white color coding. All the pixels in the image are set to 0 if they belong to the background of the play field, and to 1 if they belong to either one of the two paddles or the ball. This coding helps the network to distinguish better the key elements in the screen from the background and helps the convergence of the network.

- Transform an image to a Tensor object. This transformation is required if we want to handle images with the CUDA cores of a GPU. CUDA cores of GPUs are able to handle Tensors with more than 3x the velocity of a common CPU. Therefore, this transformation helps us exploit this functionality and speed up the training.

Examples of preprocessed game frames are show in Figure 1.

Algorithm 1: DQN with experience replay

```
Initialize replay memory $M$ and an integer $b$ for GLIE update;
Initialize policy net $Q_p$ with random samples from $\sim N(0,1)$;
Initialize target net $Q_t = Q_p$;
for episode = 1,p do
    $\epsilon = b/(b + episode)$;
    Observe first frame of the game $s_0$;
    for i = 1,I do
        With probability $\epsilon$ select a random action $a_i$;
        Otherwise draw $\Phi(s_i)$ from buffer and select $a_i = \max_a Q_p(\Phi(s_i), a, \theta_p)$;
        Execute action $a_i$ and observe reward $r_i$ and new image $s_{i+1}$;
        Push transition $T_i = (s_i, a_i, r_i, s_{i+1})$ to the buffer;
        if buffer is full then
            store transition $T_i = (\Phi(s_i), a_i, r_i, \Phi(s_{i+1}))$ in $M$;
        end
        Sample random minibatch of transitions $T_j = (\Phi(s_j), a_j, r_j, \Phi(s_{j+1}))$ from $M$;
        Set $y_i = \begin{cases} \mathrm{r}_j & \text{for terminal } \Phi(s_{j+1}) \\ \mathrm{r}_j + \gamma \max_a Q_t(\Phi(s_{j+1}), a|\theta_t) & \text{for non terminal } \Phi(s_{j+1}) \end{cases}$;
        Perform an update step for $Q_p$ on $|y_j - Q_p(\Phi(s_j), a_j|\theta_p)|$ for the entire batch;
    end
    if episode % C == 0 then
        Update $Q_t = Q_p$;
    end
end
```

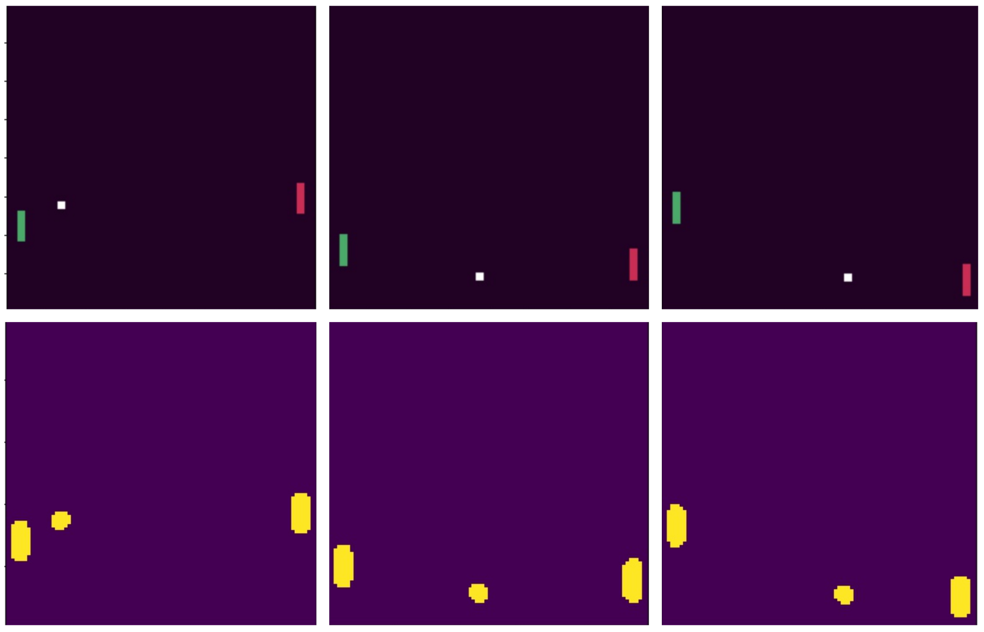
Figure 1: The first row of pictures contains 3 random observations from the game, The second line contains the same 3 frames after being preprocessed by Φ. The black and white images are blurred due to the downscaling of the original image.

2.4 Frame stacking

When we play a game like Pong, the trajectory that the ball is following or the direction in which the opponent is moving are key information that are necessary to play the game effectively. However, the single frames returned by the environment do not contain this information. In fact, a static image in a game of Pong can't tell us if the ball is moving right or left, up or down, or if the opponent is moving towards or away from the ball.

For this reason, every observation we provide to the agent is not composed by a single frame but is formed by 4 consequent frames stacked one of top of the other. This sequences provide a sense of movement and should help the agent to learn a better policy. We provide these stacks via a buffer implemented into the experience replay memory. The buffer works as follow:

- Timestep $t < 4$: The environment ϵ return a Transition T_t and pushes it to the buffer. When the agent needs to take an action, the agent consults the buffer, retrieve all the observations in the past transitions, stack them, and repeat the most recent observation until the sequence reaches a length of 4.
- Timestep $t >= 4$: The environment ϵ return a Transition T_t and pushes it to the buffer. If the buffer is full (contains 4 images), the observations are stacked and a new transition composed by the stacked frames, the most recent action and reward is pushed into the experience replay memory. When the agent needs to take an action, the agent consults the buffer, retrieve all the observations in the past transitions and stack them. Before observing the transition at time $t + 1$, the oldest transition in the

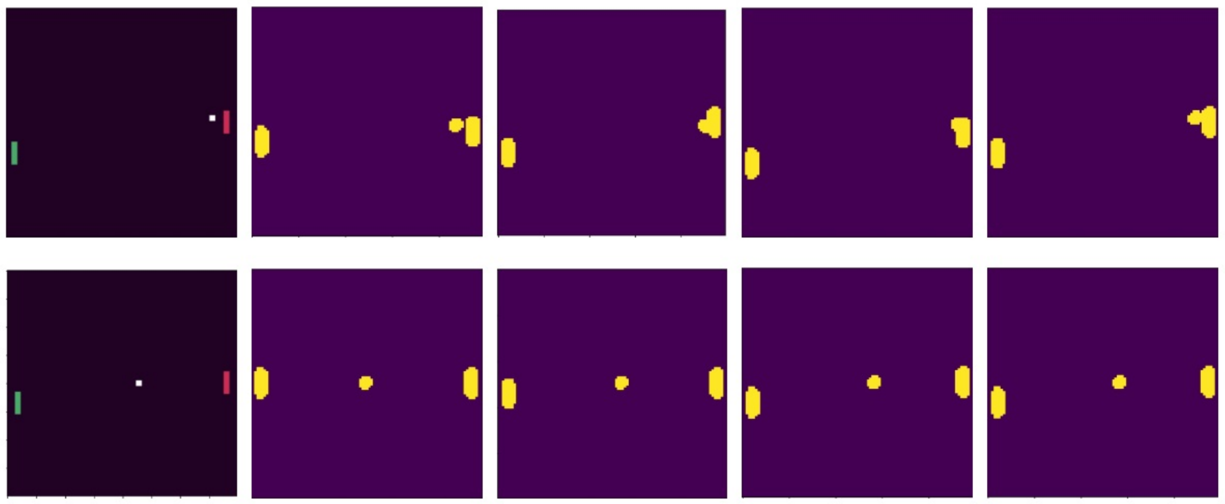
Figure 2: The first row of pictures shows an observation at timestep $t >= 4$ and the relative stacked observations retrieved from the buffer. In the stack we obtain the observed preprocessed frames at time $[t-3, t-2, t-1, t]$. The second row of pictures shows an observation at timestep $t = 3$ with the relative stacked observations retrieved from the buffer. The first 2 frames in the buffer contains the observations at time $t = 1$ and $t = 2$, the last 2 contains the same preprocessed frame at time $t = 3$

buffer is deleted.

Some examples of observations and returned stacked frames from the buffer are shown in Figure 2.

2.5 CNN architecture

We now describe the exact architecture of the neural network used for this task. The input of the neural network consists of a 4x100x100 image produced by the stacking of 4 consequent frames preprocessed by Φ. The first hidden layer convolves 16 filters with dimension 8x8, stride 4, and applies a ReLU activation function. The second hidden layer convolves 32 filters with dimension 4x4, stride 2, and applies again a ReLU activation function. The third layer is a flatten layer with no activation function that transform the previous layer's output with dimension 32x11x11 into a vector long 3872 units. The last hidden layer is a fully connected layer of 256 units and ReLu activation function. The output layer is a fully connected linear layer with a single output for each valid action, in our specific case 3. A sketch of the network structure is shown in Figure 3. The network is update through an Adam optimizer with a starting learning rate of 0.001. The weights of the network are initialized from a Gaussian distribution with mean 0 and scale 1 and are kept in a range between -0.1 and 0.1 for the entire training via pytorch clip function.

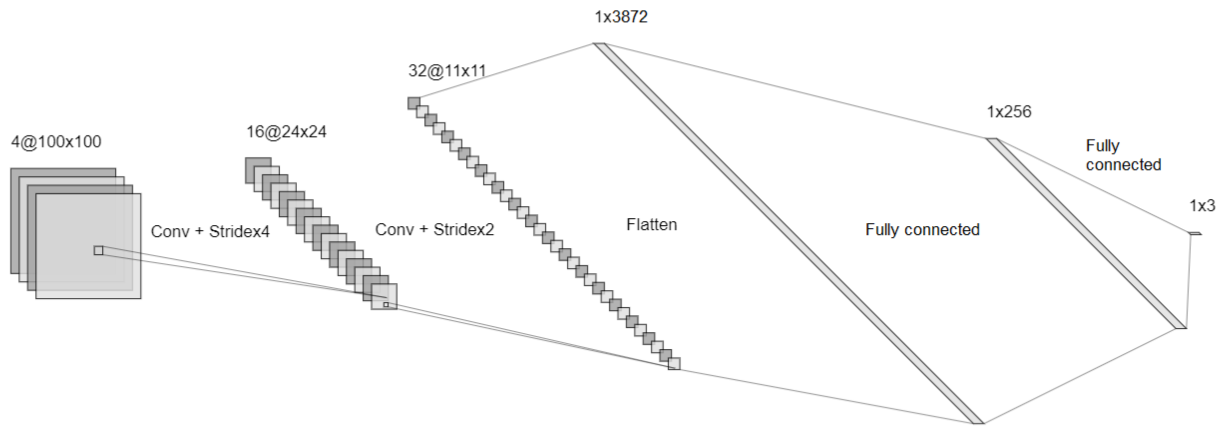

Figure 3: Sketch of the neural network used

3 Training Strategy and Performance

In the beginning, the agent has been trained to play with simpleAI for 31000 games. The models were saved periodically in order to check the performance and test against simpleAI: this ensured that we would pick the best performing model, and not just the latest one trained, which may not be the agent who wins the most.
This model, whose performance plot can be seen in Figure 4 , was able to win 73% if the games against simpleAI. The problem with this first iteration is that the agent overfitted when always playing against simpleAI, in fact when playing against itself its win-rate was only 50% and its behaviours were completely random. This was probably due to the fact that the opponent was not moving like simpleAI, and our agent relied on those movements to choose the correct action.

To solve this problem, a new type of training technique was implemented, which will be called “hybrid” for the purpose of this report. The model was now trained playing one game against simpleAI and one against itself. By using this method, the win-rate against itself was significantly improved, increasing to 70% from the 50% that we were previously achieving, losing only 2% on the win-rate against simpleAI.
The plot for this approach can be seen in Figure 5.

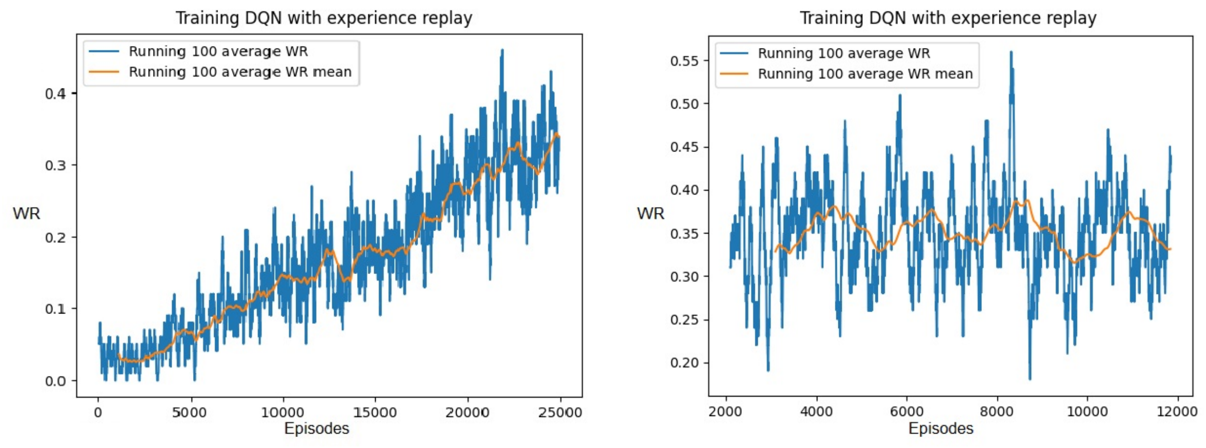

Figure 4: Plots for the first part of the training against simpleAI. It is important to notice that the win-rate is at around 0.4 during training due to our epsilon greedy being very high, thus prioritising exploration, and the score shown being an average of 100 games. Two plots are represented because the training was stopped and then started again in two different sessions.

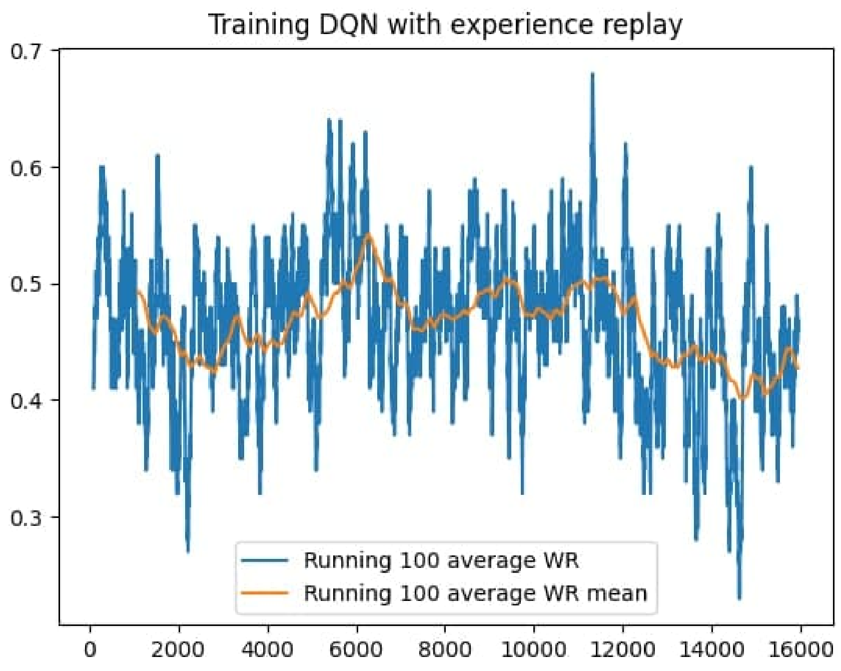

Figure 5: Training plot for the "hybrid" approach.

For the last part of the training, a technique that takes inspiration from Stochastic Weight Averaging (SWA)[2] was implemented. There are two key points that define the

SWA approach. First, it uses a modified learning rate schedule so that Adam, our optimizer, continues to bounce around the optimum and explore diverse models instead of simply converging to a single solution. And, secondly, SWA performs, at training time, an equal average of the weights of each training's checkpoint. This novel approach avoids overfitting the agent and other common training errors, and has shown to significantly improve the performance of different reinforcement learning algorithms [3].
A new model was therefore created by taking the ten best weights obtained during the "hybrid" step and averaging them to create the new model. This increased the win-rate against simpleAI to 78%, and against itself to 75%.

Table 1 shows all the most significant hyperparamenters used during all the training phases of the agent.

All the fine tuning described has the goal to improve how the agent plays against different opponents, and not just simpleAI, which hopefully will improve our results in the final tournament.

4 Possible improvements

Significant improvements could be made by increasing the resolution of the preprocessed observations. In some cases, the ball is so close to the agent that it is difficult to determine its exact position due to the blurriness of the image. By increasing the resolution, we would lose the benefits gained by doing it in the preprocessing phase, thus increasing the training time.
Another weakness that we noticed during testing happens when the ball is far from the position of the paddle or when the ball is going too fast. When this situation occurs, our agent is either not able to reach the ball or simply does not move. This could probably be improved by fine-tuning the reward function so it is able to handle these edge cases.
Other approaches were considered before we decided to implement our current solution.
Policy Gradients methods have been shown to be more performant than Q Learning when tuned well with an explicit policy and a principled approach that directly optimizes the expected reward [4]. This approach would also require less memory and computational power, whilst not limiting us to off-policy learning algorithms that can only update from data generated by an older policy. More novel implementations also asynchronously execute multiple agents in parallel on multiple instances of the environment, which, most notably, decorrelates the agents' data allowing us to solve this problem using both on-policy RL algorithms, such as Sarsa and actor-critic, and off-policy, such as Q-learning.

5 Conclusion

Before starting this project we discussed different approaches that could have solved the problem, and our choice fell on Deep Q-learning for the reasons described in this report.
Our agent does not follow a particular strategy to win, and mainly tries not to lose against its opponent, so with more time available for training and fine-tuning our agent could improve

Optimizer	Adam
Learning Rate	0.001 for the first training phase and 0.0001 for the following ones
Batch Size	128
Memory Capacity	40000 Transitions
Buffer Capacity	4
Gamma	0.98
Glie_a	10000
Minimum Epsilon	0.1
C	50

Table 1: Hyperparameters

even more. The knowledge that we acquired while developing this project could be used to implement other approached stated in Section 4.
However, Deep Q-learning is very effective for training an agent that plays Pong, and with our implementation the win-rate reaches 78% against simpleAI, and 75% against itself. This value will be high enough to play against other agents an possibly achieve a high position in the final ranking.

References

[1] D. S. A. G. I. A. D. W. M. R. Volodymyr Mnih, Koray Kavukcuoglu, "Playing atari with deep reinforcement learning," tech. rep., DeepMind Technologies, 2013.

[2] A. G. W. Pavel Izmailov and V. Queneneville-Belair, "Pytorch 1.6 now includes stochastic weight averaging," August 2018. [Online; https://pytorch.org/blog/pytorch-1.6-now-includes-stochastic-weight-averaging/].

[3] B. A. D. P. T. G. P. S. D. V. A. G. W. Evgenii Nikishin, Pavel Izmailov, "Improving stability in deep reinforcement learning with weight averaging," tech. rep., National Research University Higher School of Economics, Cornell University, Samsung-HSE Laboratory, Samsung AI Center in Moscow, 2018.

[4] M. M. A. G. T. H. T. P. L. D. S. K. K. Volodymyr Mnih, Adrià Puigdomènech Badia, "Asynchronous methods for deep reinforcement learning," tech. rep., Google DeepMind, Montreal Institute for Learning Algorithms (MILA), University of Montreal, 2016.