Reinforcement Learning

Eugeniu Vezeteu, Oliver Horst
Aalto University - Electrical Engineering
ESPOO, Finland
vezeteu.vezeteu@aalto.fi,oliver.horst@aalto.fi

ABSTRACT

This is the final year project for the Reinforcement Learning course, at Aalto University
We designed an reinforcement learning agent to learn to play Pong directly from pixels. After several hours of training, our method is able to outperform the existing Simple AI agent.

INTRODUCTION

Reinforcement learning is a branch of Machine learning that deals with training agents who interact with the environment to make a sequence of decisions and maximizing the total reward. The agent learns to accomplish a goal in an uncertain and complex environment. It uses trial and error to find a solution to the problem and receives rewards or penalties for each action.
Our environment is a Pong game, where 2 players play against each other. The player score if the ball passes the opponent and lose otherwise. A sceenshot from the game is presented in Figure 1.

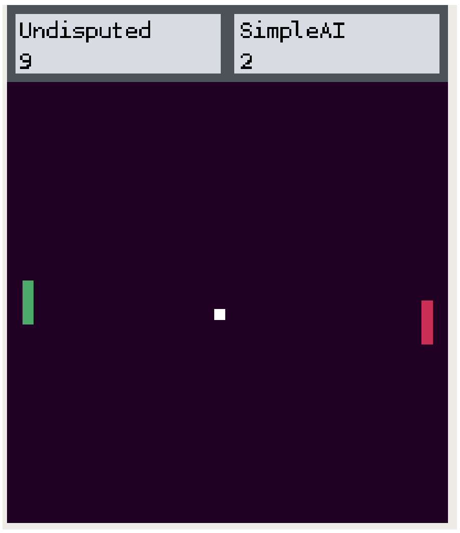

Figure 1: Pong - Environment

Our agent is the green one, and it is playing against provided SimpleAi.

- The state space is continuous and it's a frame of [200x200x3] dimension.
- The action space is descrete, 3 actions: 1,2 and 0 for (go up, move down and stay).
- The environment return 10 reward for winning the game, -10 for loosing and 0 otherwise.

The goal of the agent is to learn to play the game from pixels.
One approach to solve the problem is the Policy Gradient method presented in the following chapters. The reasons for choosing this method are the following:

(1) It is applied for large and continuous state space
(2) Can learn a stochastic policy
(3) Its a relatively easier algorithm (compared to AC family) and according to Assignment 6 Figure 1 "Training performance for different algorithms" from the RL course, it outperformed the more complex AC method in the CartPole environment. As pong is also a relatively simple environment, it is expected to be a comparable.

REVIEW OF EXTERNAL SOURCES

To better understand the problem and how to learn from pixels, we took inspiration from Karpathy's blog [1] and [3]. As the purpose of this project is not model selection, but the use of a model for an agent to learn to perform a task, we borrowed the policy architecture from [6]. The agent's design was inspired from Reinforcement Learning lessons at Aalto University, [2] course assignment 5 and Sutton, Richard S and Barto, Andrew G book [5]. We used Google Colab GPU to train the agent, according to the code hint given in the project report guide.

DESIGN OF THE AGENT ARCHITECTURE

The input to the network in an image frame of size $[200x200x3]$, which will in later steps be reduced to a $[50x50x1]$ image frame to reduce the parameter space. To process the temporal dependencies we decided to subtract each 2 consecutive frames and pass the difference as an additional input to the network. The policy architecture is presented in Figure 2, the input is represented by $[2 \times 50 \times 50]$, where we have stacked 2 frames of size $[50 \times 50]$, the first channel is the original frame downsized, and the second channel is the $\Delta frame$, where $\Delta frame$ is $frame_{i+1} - frame_i$. We used 3 convolutional layers to extract the image features. For each convolutional layer we used kernels of $[5x5]$ dimension. We learn the features by passing them through fully connected layer and then use the last fully connected layer with 3 nodes, for each of the actions. For each convolutional layer we used batch normalization to make the neural network training faster and more stable by normalizing the previous layer , re-centering and re-scaling it, as described in the paper [4].
To introduce non-linearity we used ReLU activation function ($z = max(0, h)$) for all layers, except the last one. On the last layer, we used softmax to output the probability distribution for each action.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

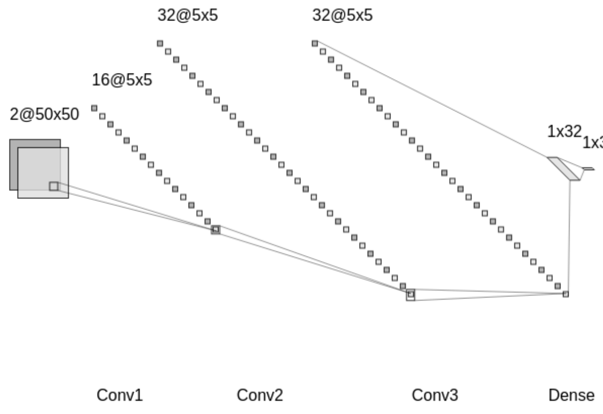

Figure 2: Policy architecture

We computed the discounted reward for each episode using:

$$G = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

and used it to update the policy parameters by:

$$\theta = \theta + \alpha * G * \nabla ln\pi(A_t|S_t, \theta)$$

TRAINING METHODOLOGY

We want to inform the reader that the training of the agent lasted in total 24-25 hours, but these hours were divided into a few days, because Google Colab automatically closes the session after some time. We apologize for not presenting the training results in a single plot, but these are presented and evaluated as we obtained them.
Besides the number of time-stamps and the rewards, we computed the win rate in following way:

$$win\_rate = \begin{cases} +1 \quad if \quad the \quad agent \quad won \quad the \quad game \\ \qquad 0 \quad otherwise \end{cases}$$

The environment returns reward of -10 for loosing and 10 for winning. We decided to change it to -1 for loosing and 1 for winning, 0 for staying alive. We decided to change the reward function for stability purposes (it is easier to work with smaller numbers (-1,0,1) instead of (-10,0,10)).
We apply the preprocessing step to the frame, before passing it to the policy. Initial frame dimension is [200x200x3], we apply the following steps:

(1) normalize the frame: *frame = frame/255.*
(2) downsize to 50 pixels and take the average on the last channel: $image = image[:: 4, :: 4].mean(axis = -1)$, after this step the frame has dimension [50x50x1]
(3) using the previous observation, we stack two frames: the current observation from (2) and the difference between this and the previously preprocessed image, resulting in a [50x50x2] dimensional frame

This frame now consists of the most important information about the environment, both in terms of locations (1st channel) and velocities (directions, 2nd channel).
Initially, a slightly different approach for the preprocessing had been chosen, in which the frame was downsized to a [100x100x1] image and segmented and processed in the following way:

(1) the left column of the image, containing the player's pedal was left untouched, giving the absolute position
(2) the middle column, containing the ball, was processed to contain the difference over the last timestep, giving the direction and velocity of the ball
(3) the right column, containing the opponent's pedal, was considered less important and entirely removed

Using this technique, a final winrate of 84% against the Simple AI agent was achieved. This has been dismissed however, in order to present a more general algorithm. This trained agent will however be used as an additional opponent during the training of the general agent. Additionally, the policy
We used the Monte-Carlo approach and update the policy after each episode. We used *RMSprop* optimizer with learning rates: [1e-4,1e-5,1e-6] and started with training our agent against the Simple AI agent. After reaching a certain winrate, the training opponent has been alternated between Simple AI and the other, previously presented, well performing agent in order to generalize the agent's techniques. This is important, as most opponents will behave quite differently than the Simple AI agent.

EVALUATION OF RESULTS

The result of the first training phase are presented in Figure 3. After this initial training, the agent was able to achieve 0.30 winrate against Simple AI.

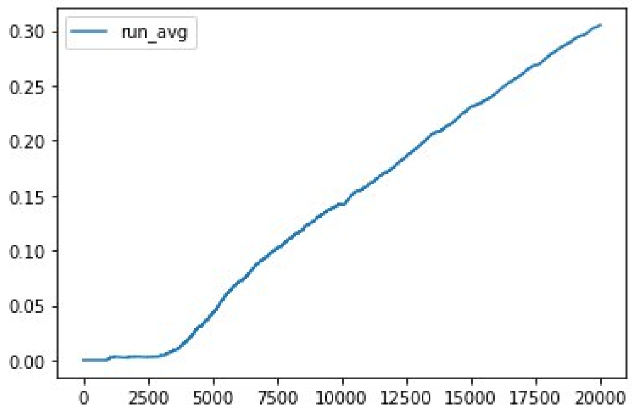

Figure 3: Training phase 1, win rate

In the second training phase we used the model trained above with win rate 0.30 and decreased the learning rate to 1e-5. The result is presented in Figure 4, the agent achieved 0.717 win rate. In the following step we decreased again the learning rate to 1e-6 and let the agent train again, this time training against the previously presented, more specialized, agent. The result is presented in Figure 5. At the end of this training, the agent's performance plateaued at a winrate of 73.8% when competing against the Simple AI algorithm, while it could win 42.4% of the games against the more specialized agent.

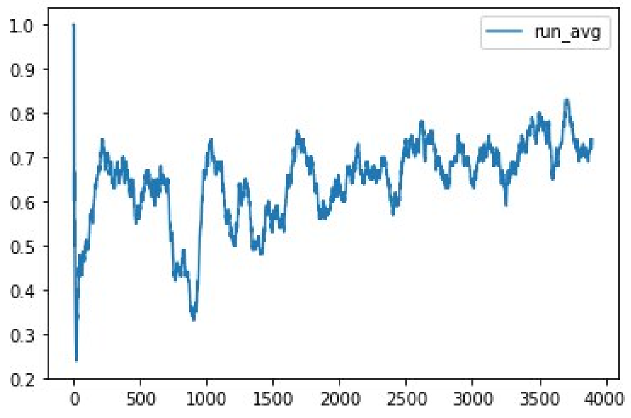

Figure 4: Training phase 2, win rate

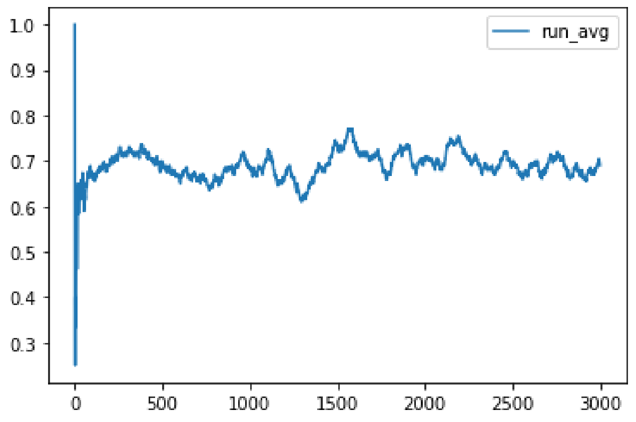

Figure 5: Training phase 3, win rate

In the Figure 6 we present the average timestamps for each game, using the last trained model.

This final model (after phase 3) has been selected and it's performance will be assessed in the following.

In order to visualize the testing of the agent against the Simple AI agent, Figure 7 displays the result of 1000 games played: Won games are displayed in green, lost games in red. In five independent test runs, both over 1000 games, the agent was able to achieve an average win-rate of 72.5% with a standard deviation of 1.1%. As a result of playing 1000 games, we saved a 1000 array and reshaped it to $[40 \times 25]$ to display it Figure 7. The Figure 7 doesn't have X and Y axis since it is a 1000 games result visualized as image. The green pixels correspond to win game and yellow to lose.

CONCLUSIONS.

A reinforcement learning agent based on policy gradient has been presented to outperform a simple, "following" agent in most games. The low deviation of won games during the test runs proves the agent's performance to be very stable and suggest the policy gradient method to be well suited for the task of playing a simple game such as Pong.

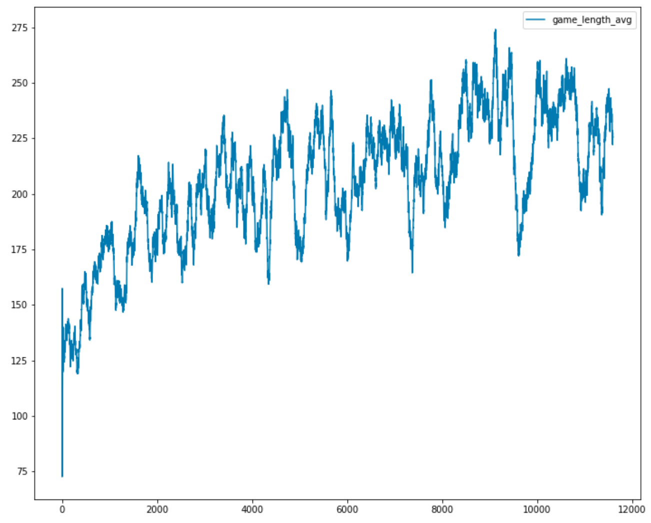

Figure 6: The last training phase, avg timestamps

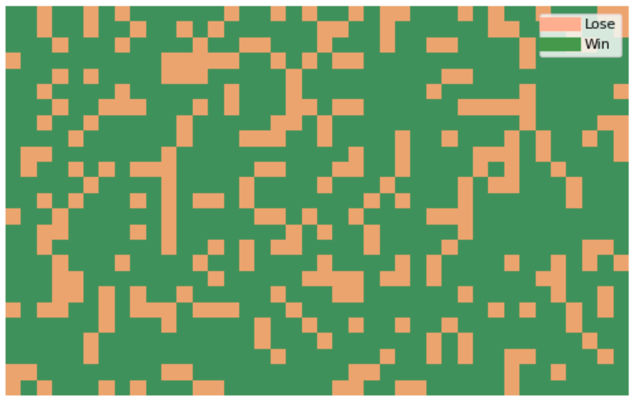

Figure 7: 1000 games against Simple AI (green=win, red=lost)

For future works, it will be suggested to test the performance of the presented algorithm on other, slightly less simple Atari games such as Breakout or Space Invaders in order to test our hypothesis of having implemented a generalized agent, but also test how far this simple policy gradient method is able to go on more complex environments.

REFERENCES

[1] Andrej Karpathy. 2016. *Deep Reinforcement Learning: Pong from Pixels.*
[2] Ville Kyrki. 2020. *Lecture slides - Lecture 5, slide 19.* Aalto University.
[3] Machine Learning Projects. 2018. *Pong Game.*
[4] Christian Szegedy Sergey Ioffe. 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.*
[5] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.
[6] Álvaro Barbero Jiménez. [n.d.]. *deeprl-pong.* https://github.com/albarji/deeprl-pong