ELEC-E8125 - Reinforcement learning

Final Project Report

Author:
Sheetal Borar *(915263)*
Hossein Firooz *(903929)*

Lecturer:
Prof. Ville Kyrki

December 7, 2020

Contents

1 Introduction

We aim to describe our solution for playing pong with pixels using reinforcement learning in this report.

1.1 Wimblepong environment

The *wimblepong* environment is a custom OpenAI Gym environment which consists of two paddles and a ball as shows in Figure 1. Each paddle is controlled by one of the players. Every time the ball passed the paddle, other player gets a score. The goal of this project is to create an agent using reinforcement learning to maximize the winning rate.
We used the *WimblepongVisualSimpleAI-v0* version of the environment. This version returns $200 \times 200 \times 3$ image which respectively shows the height, width and colour depth of the image. In this environment, there are three possible actions: Move up, Move Down, and Stay.

Figure 1: Wimblepong Environment

2 Review of external sources

One of the most popular works in Reinforcement learning in the recent times is [1], which produced a common deep network architecture which could learn to play 49 different games. We referred to the project for some preprocessing steps like making the image gray scale, reducing its size and stacking the images together to provide motion information to the network.

We also referred to [2] and [3] to create the final network architecture using convolutional and batch normalization layers. These papers describe the benefits of using CNNs in image based reinforcement learning task and the improvement in performance stability

obtained through batch normalization.

We used Actor Critic method for our agent which is fully described in [4]. We also used Entropy loss as part of our objective function. This idea was described in [5] and was used in recent actor critic methods like [6].

We got ideas from OpenAI Baselines [7] Tensorflow implementation, some of the hyperparameters were borrowed as well. However we read some best practices for the hyperparameters in [8]. In addition, we referred to [9] for a better understanding of gradient clipping.

3 Agent 47: Design Architecture

This section explains the design techniques we take to design our agent: **Agent 47**.

3.1 Input Preprocessing

We do three things to preprocess the input before feeding it to the network:

- Gray Style
- Down Sampling
- Stacking Frames

3.1.1 Gray Style

Pong is a simple game and beside the paddles and the ball, there are not very useful information in each scene. In addition, paddles only goes up and down and the x position of the paddle is sufficient for recognizing which paddle is ours and which paddle belongs to the opponent. Therefore by reducing the color channel, we use a simpler version of input image without using useful information. This technique was also used in [1]. We take the mean of the image along the colour channel to make it gray style therefore the $200 \times 200 \times 3$ input becomes a 200×200 image.

3.1.2 Down Sampling

W reduce the width and height of the image by factor of two. This gives us an image of size $100 \times 100 \times 1$. This down sampling would create a smaller image for our input which means we have smaller number of weight in our NNs to train which causes a faster learning process. This technique was also used in [1].

3.1.3 Stacking Frames

In this game, the ball is moving around and feeding only one image to the network doesn't provide the information about the movement of the ball. Therefore we stack two images together to use as our input. This gives us the preprocessed input of size $100 \times 100 \times 2$, ready to be passed through the network. In the original paper [1], they used four images instead of two. As the time between frames are consistent and the velocity of the ball doesn't change until it hits one of the paddles, stacking two frames have the sufficient information about the ball location, direction and velocity.

3.2 Advantage Actor Critic

Our agent is a modified version of Actor Critic called Advantage Actor Critic or A2C [4]. We used actor critic because the mapping between state to action seems quite intuitive in this problem. Apart from that the state space is continuous, making it discrete to use other approaches like DQN lead to worse performance. Last but not least, A2C has a faster convergence than DQN.

3.3 Neural Network Architecture

As the observation of the agent is a raw image, we are using a deep Convolutional Neural Network (CNN) [10] to represent the agent's policy. Using CNNs for policy representation is very popular in Reinforcement Learning problems. Using such networks had great results in playing Atari games [2]. Therefore we choose CNNs for our agent.
Our first architecture design was based on [1] which achieves a human level score on playing various Atari games. This structure consist of three CNN layers, each followed by a ReLU layer.
Although using the same structure works, our experiments show using Batch Normalization would speedup the learning process as shown in Figure 2. Our main intuition behind using Batch Normalization was to reduce the dependency to the initialization weights [3] : In Reinforcement Learning problems, NN initialization has a big impact on the learning outcome. Using Batch Normalization would reduce this dependency. Therefore, we used the default initialization of weights that PyTorch provides.

Figure 3 shows the different stages of the policy network. As we are using A2C, there is actor part with the Softmax layer learning the action probabilities and there is the critic part learning the state-values.

3.4 Optimizer

As described in [6] we used RMSprob for our optimizer with the exact same $lr = 7e - 4$, $\alpha = 0.99$, and $\epsilon = 1e - 5$. However for finetuning our agent, we used a smaller learning rate of $lr = 1e - 5$. More on our fine-tuning in 3.5.3.

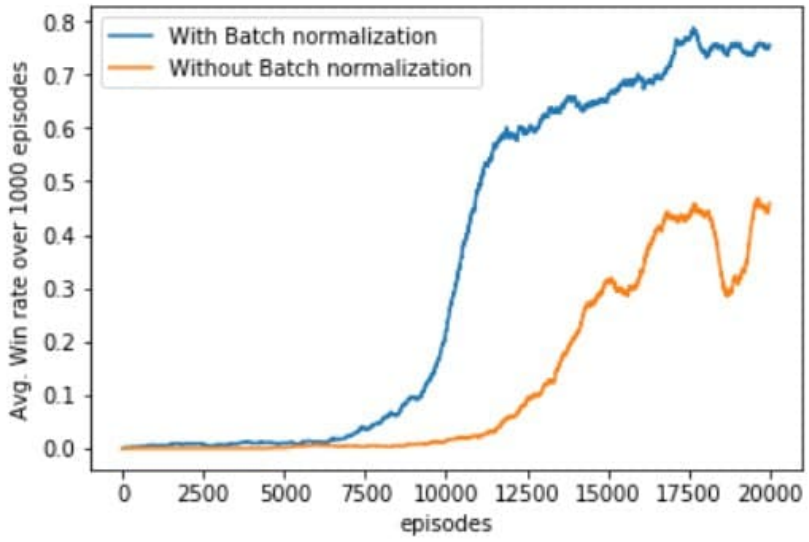

Figure 2: Effect of Batch Normalization

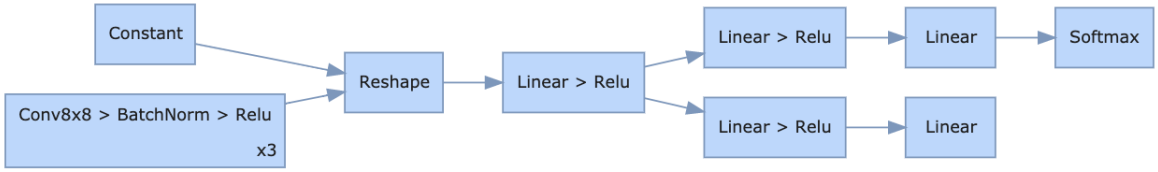

Figure 3: Architecture of the network

3.5 Loss Function

With calculating the discounted rewards R_t at the end of each episode, we calculate the advantage of each action by using $A_t = R_t - V(s_t, \theta)$ where θ is the network parameters. We then use three loss objectives:

3.5.1 Actor Loss

Which is calculated with $\nabla_\theta ln\pi(a_t|s_t, \theta))A_t$ [7]. This is based on using state-value as the baseline to create the advantage function.

3.5.2 Critic Loss

As we are using the state-values in advantage function, we need to optimize the critic part as well. We are using $\epsilon_c \nabla A_t^2$ for the critic loss where ϵ_c indicated the strength of the critic loss term. We find that using a smaller than used in [6] would result a reliable and constantly improving process. We borrow this from OpenAI Baseline implementation [7].

3.5.3 Entropy Loss

We also found that adding the entropy maximization term of the policy to the loss function improves exploration by encouraging the agent to find many paths to reach the goal, rather than premature convergence to sub-optimal policies. Before adding this term, we noticed our agents would learn to go in only one direction, which means that in some cases the agent found going in 1 direction improves the reward and it converged on that behaviour. Entropy maximization would prevent this behaviour by discouraging extreme difference in action probabilities. This technique was originally introduced in [5] and it was also used in [6]. By using entropy of the action (policy) distribution, we add entropy loss to our objective function: $\epsilon_e \nabla_\theta E(\pi(s_t, \theta))$ where ϵ_e indicated the strength of the entropy loss term. Figure 4 shows the effect of entropy loss on our training.
Although both OpenAI baseline implementation [7] and this study on on-policy reinforcement learning [8] used a constant ϵ_e, We find out reducing it overtime would speed up the training progress because it is encouraging exploration at the beginning of the learning process and preventing the exploration after a while. As the main intuition behind using entropy is to encourage explorations, we reduce ϵ_e slowly to prevent the agent from too much exploration. We start the training with $\epsilon_e = 0.1$ and divide it by 10 every 5000 episodes.

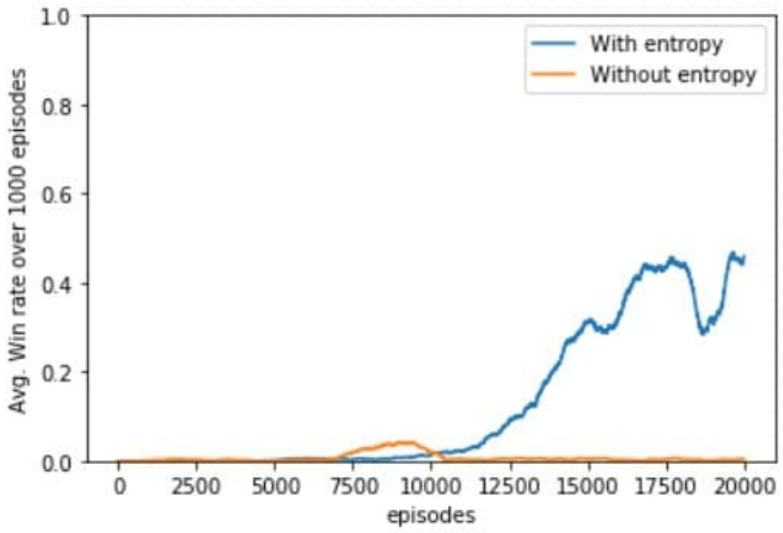

Figure 4: Effect of Entropy Loss

3.6 Gradient Clipping

Gradient clipping is a technique used to prevent exploding gradients. We used this technique with the maximum gradient of 0.5 which we borrowed from [7]. Whenever the gradient becomes to big, it can be re-scaled. [9] shows that using gradient clipping would accelerate the training process.

4 Training methodology

We used `train.py` to train our agent against the SimpleAI. Figure 5 shows this training process. Our baseline paper [1] trained the agents for 40M frames. As they stack 4 images for every input and we are stacking two images, we decided to train our model for ≈20M frames. This training was done on a `P5000` GPU for around 48 hours.
At the end of training, Agent 47 had a of winning rate of **85%** against SimpleAI.

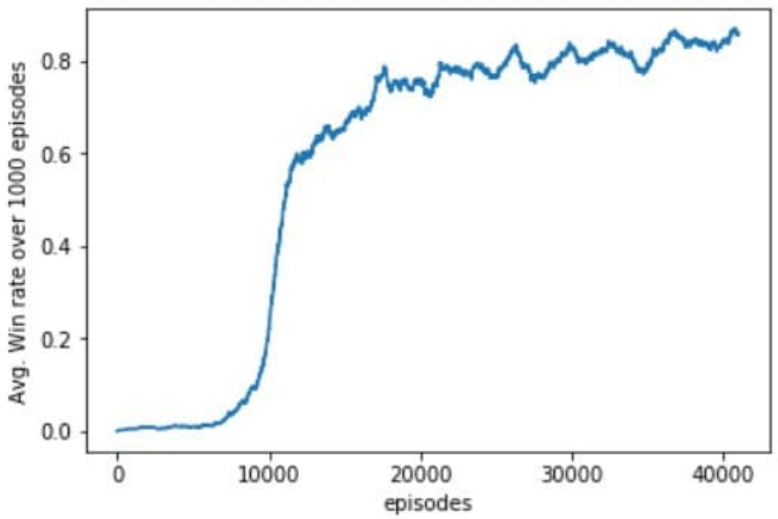

Figure 5: Training Process

4.1 Fine Tuning

Training Agent 47 for more episodes, didn't increase the winning rate. That's why we decided to fine tune our agent using another approach. We used three different approaches for fine tuning our agent:

- Smaller Learning Rate
- Training against Itself
- Ignoring the opponent paddle
- Ensemble Training

4.1.1 Smaller Learning Rate

Looking at Figure 5 we observe that after 30000 episodes, it seems the learning is very slow. Therefore we decided to fine tune our agent manually. We load the learnt model and chose a smaller learning rate of $7e-5$, $\epsilon_e = 1e-2$ and $\epsilon_c = 0.25$. We hyper-tuned our

agent against SimpleAI and we were able to achieve 92% of winning rate instead of 85%. However, this training had a side effect: Our agent were **overfitted** to the SimpleAI behavior and act poorly against itself.

4.1.2 Training against itself

Our second try was to train the agent against itself rather than SimpleAI. Although Agent 47 learned an optimal policy and was able to achieve 96% winning rate against the pre-finetuned model, but the overfitting problem was still here. Instead of SimpleAI, our agent were **overfitted** to it's own behavior and were loosing against SimpleAI 50% of the times.

4.1.3 Ignoring the opponent paddle

Our agent was overfitted in either of two previous approaches. It means our agent was mainly reacting to the other paddle instead of just considering the position of the ball. We decided to try another approach were we removed the other paddle in the preprocessing: We removed the first 12 pixels from the width. Therefore the image would be 200x188 instead of 200x200 and doesn't contain the other paddle. By using this approach, we were trying to teach Agent 47 how to react only to the ball and not taking account for the opponent paddle position. Although the opponent paddle is also important, Our intuition was that the ball position is a good proxy for getting the other paddle position as the opponent is reacting to the ball position.
This approach didn't result in an optimum behavior. This is because both the position of the ball and the other paddle are important for learning to beat the other agent and having only the position of the ball would loose some useful information.

4.1.4 Ensemble Training

To avoid overfitting to the behavior of the opponent, we trained our agent against an ensemble of opponents: We switched the opponent every 10 episodes between SimpleAI and Agent 47 pre-finetuned model. This way our agent learned how to remain flexible.
We can observe in Figure 6, that the agent unlearned the overfitted behaviour, which caused the winning rate to decrease to 60% and then learnt a flexible policy that works against both the opponents.
This fine-tuning took 12 hours on `P5000` GPU. The code is provided in `train_multi.py`

5 Evaluation of results

By fine-tuning our model using Ensemble Training, we were able to obtain a **98%** winning rate against simple AI on average using `mass_test_simple_ai.py`. In addition, we were able to beat pre-fine-tuned model with a **71%** win rate with using `epic_battle_royale.py`.

Table 1 shows a summary of all the hyper-parameters we used to train Agent 47

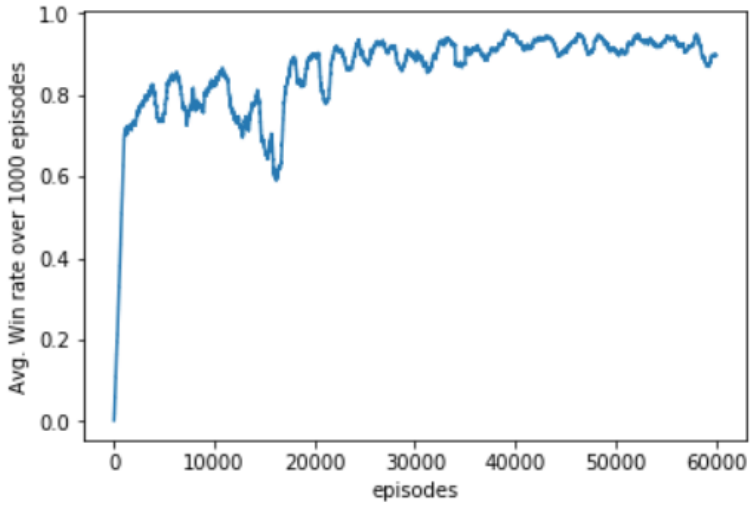

Figure 6: Training plot: Fine-tuning with 2 opponents

HyperParameter	**Training Value**	**Fine-Tuning Value**
Gamma	0.9	
ϵ_c	0.5	0.25
ϵ_e	1e-2	1e-3
Maximum Gradient	0.5	
Optimizer	RMSProb	
RMSProb LR	7e-4	7e-5
RMSProb ϵ	1e-5	
RMSProb α	0.99	

Table 1: Hyperparameters

6 Conclusions

In this project we created an agent which were able to learn the game of pong by just looking at the raw images. This project clearly shows the power of reinforcement learning. However during this project, we were facing with some hard issues:

- **Sample Inefficiency** We needed ≈21M Frames of the game to teach (and fine-tune) our agent. This clearly shows how sample inefficient are RL approaches.
- **Training Time** We used a powerful GPU for training our agent and it took around 60 hours. Considering the fact that we didn't use a very deep neural network to represent the policy, training time is coming mostly from the simulation: Simulating every action in the game takes time.
- **Local Optima is Be Hard To Escape** Without fine-tunning, our agent was stuck

at winning rate of 85%. We had to fine-tune it with different approaches to escape this local optima.

- **Overfitting** Fine-tune our agent against itself and the SimpleAI cause the agent to overfit its behavior to the other paddle movement. This means although our agent was doing good against the other opponent, but it cannot generalize how to win the game.
- **Sensitivity to HyperParameters** Changing almost every hyperparameter would completely change the final results. This shows the sensitivity of the hyperparametrs.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[5] R. J. Williams and J. Peng, "Function optimization using connectionist reinforcement learning algorithms," *Connection Science*, vol. 3, no. 3, pp. 241–268, 1991.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[7] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[8] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," *arXiv preprint arXiv:2006.05990*, 2020.

[9] J. Zhang, T. He, S. Sra, and A. Jadbabaie, "Why gradient clipping accelerates training: A theoretical justification for adaptivity," *arXiv preprint arXiv:1905.11881*, 2019.

[10] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*, pp. 267–285, Springer, 1982.