Aalto University
School of Science

ELEC-E8125 Project work: Pong42

REINFORCEMENT LEARNING

Mattia Zoffoli 802716,
Xinyi Tu 785419,

MONDAY 9TH DECEMBER, 2019

1. Introduction

The project work for the course (ELEC-E8125 - Reinforcement learning, 09.09.2019-04.12.2019, Aalto University) is about implementing a reinforcement learning agent that can play the game of Pong from pixels. In this environment, the agent receives as input the raw frames from the game, controls one paddle and can take one of three actions: moving up or down, or staying in place.

2. Review of External Sources

Aside from lecture material and code provided for the exercises, we used the following additional resources:

- **Deep Q Networks (DQN)** [1]: This paper presents the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using RL. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. Inspired by this paper, we employed a simple DQN algorithm as shown in section 3.4.

- **Proximal Policy Optimization (PPO)** [2]: This paper proposes a new family of policy gradient methods for RL, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform one gradient update per data sample, it proposed a novel objective function that enables multiple epochs of minibatch updates. This new method, proximal policy optimization (PPO), have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Based on this paper, we implemented our actor-critic algorithm with proximal policy optimization (PPO) as shown in section 3.1.

- **PPO Hyperparameters and Ranges** [3]: This post explains the hyperparameters and their ranges as used in the small number examples the auther gathered. Based on the recommended hyperparameter settings, we conducted several experiments as shown in section 3.3.

- **PPO Implementation** [4]: This Github folder contains an implementation of a

PPO algorithm applied to Policy Gradient in the Pong Environment. We took inspiration from this code to perfect some of our design choices.

3. Approach and Method

This paragraph will present the algorithm we chose, together with the reasons for our design choices, the values of the hyperparameters we chose, and present some alternative approaches we explored.

3.1 Algorithm Overview

The algorithm chosen for the final submission has been **Proximal Policy Optimization (PPO) in Actor-Critic style**. At each iteration, each of N actors within each episode collects varied t timesteps of data until the episode is over, then we construct the surrogate loss on these Nt timesteps of data, and optimizer it with minibatch Adam, for K epochs. Slightly different from what was proposed in the PPO paper [2], our method does not employ fixed-length trajectory segments and parallel actors. The pseudocode of the algorithm we conducted is shown below.

Algorithm 1 PPO, Actor-Critic Style

for iteration = 1 ,2, ... **do**
  # *Gather Experience:*
  for episode = 1 ,2, ... , Horizon **do**
    for timestep = 1 ,2, ... **do**
      Forward Actor policy network based on previous observations
      Step the environment and get the new observations and rewards
      Store transition results of each timestep
    end for, when one episode is over (*done* == *True*)
  end for
  # *PPO Update:*
  for epoch in *k_epochs* **do**
    Sample a random Batch_size of the stored data
    Forward Critic policy network based on batch actions and states
    Optimize surrogate loss wrt. network weights
    Pass new weights to old policy network
  end for
end for

3.2 Design Choices

The following design choices characterize our model:

- **Network Architecture**: two fully connected layers, with an hidden layer of 512

neurons. Only the first layer is shared between the actor and critic networks, which compute respectively the action logits and the values.

Initially, we used some convolutional layers to process the frames before the fully-connected layers, however, we noticed the learning was faster without convolutions, taking inspiration from the network architecture used in the code of resource [4].

- **Preprocessing**: we preprocess the frames eliminating the background in a similar way to that used in the code of resource [4], by setting the background as 0 and the rest as 1. Additionally, we grayscale, downsample the frames by a factor of 2 or 3 (testing both factors didn't change the learning curve significantly) and flatten the observation array.

- **Memory**: our model stores the previous observation, and stacks it with the present one, feeding to the network a stacked object made of two preprocessed observations.

- **Reward Function**: initially we tried to train our agent with a reward function which gave reward to the agent proportionally to the length of each episode, because we wanted the agent to learn to play longer, because a longer game increases the chances of winning. However, this method achieved worse results with the same model, therefore we discarded it, and trained our final model with the **default reward function**. It would have been interesting to try training with a reward function which penalize the agent for being vertically distant from the ball (using an additional network trained with supervised learning to extract the positions of each element), however this process would have been computationally not efficient, therefore we didn't try it.

- **Weight Initialization**: we initialize the weights as slightly different from zero. Our model was already learning without this technique, but when we started using it, the initial learning curve improved significantly.

- **Batch Update**: our model initially had memory problems when we used the full experience to update the network, therefore, taking inspiration from the code of resource [4], we started sample a random 30% of the data available before each epoch of parameters update.

- **Action Space Dimension**: after some experiments we decided to reduce our action space to just UP or DOWN, eliminating the possibility of the agent staying in place. This makes the movement of our agent less fluent, but it ensures a faster training due to a reduced dimension of the output layer of our network.

- **Advantage Estimation with Values as Baseline**: the estimated advantage function for the new policy is calculated in the form of the expected rewards minus a baseline, which is the values returned by the policy network' forward step in the PPO update.

- **Clipped Surrogate Objective**: without a constraint, maximization of "surrogate" objective loss would lead to an excessively very large policy update. As proposed in the PPO paper [2], we employed the clipped surrogate objective, which takes the minimum of the clipped and unclipped objectives, so the final objective is a lower bound on the unclipped objective. Inside the minimum function, the clipped objective term modifies the surrogate objective by recalibrating with the

probability ratio between the new and the old policy, which removes the incentive for moving ratio outside of the interval [1- ϵ, 1+ϵ].

- **State-Value Function Loss Term**: As explained in the PPO paper [2], with a neural network architecture that shares parameters between the policy and value function, we must use a loss function that combines the policy surrogate and a value function error term. The later is employed in the squared-error loss form.
- **Entropy Regularization**: A policy has maximum entropy when all policies are equally likely and minimum when the one action probability of the policy is dominant. We further augment the objective loss function described above by adding the entropy bonus term to loss function. It helps prevent premature convergence of one action probability dominating the policy and stopping exploration.
- **Optimizer**: we choose Adam with default beta parameters as our optimizer, because it is generally considered the best choice, and we considered this parameter less relevant than others.

3.3 Hyperparameters Choice

PPO is a robust algorithm and it can achieve good results without optimal hyperparameters initializations, meaning the same algorithm can be used for a variety of different RL Tasks. However, proper hyperparameter initialization and search can still lead to improved results.

The following list presents all the values we chose for our hyperparameters, which were inspired by resource [3]

- **Updating Horizon**: Initially we tried using a small number of timesteps between each network update, then we thought the agent would need to win several times before each update, therefore we finally decided to update the network using an horizon of **200 episodes** of experience.
- **Training Epochs**: initially we used a small number of training epochs (5), however, introducing batch update we decided to raise the number of epochs to **12** in order to use each episode more than once for every updating step, to increase sample-efficiency.
- **Learning Rate**: the default learning rate we used for our experiments was 0.001, however when we introduced batch update and increased the number of training epochs, we reduced it to **0.0003** in order to reduce the size of each updating step.
- **Discount Rate**: our chosen discount rate (gamma) has been **0.99**. We tried using a lower value (0.97) in order to make the agent try to win quickly by bouncing the ball on the side of the paddle rather than in its centre, however we noticed that the learning was less effective, and ultimately our model learned to try to bounce the ball diagonally even with a higher discount rate.
- **Hidden Layer Size**: our network has an **unique hidden layer of 512 units**. We also experimented with 256 and 1024 units.

- **Clipping Parameter Epsilon**: as discussed above, policy performance can collapse dramatically and never recover if the policy is updated in too large a step. Therefore a surrogate loss function is employed to keep the step from the old policy to the new policy within a safe range. We experiment with common value **0.1** and **0.2** as clipping parameter epsilon.
- **Value Function Error Coefficient**: the value function error coefficient is multiplied by value function squared-error loss as a term in the objective loss function. We used the most common value **0.5**.
- **Entropy Coefficient**: the entropy coefficient is multiplied by the maximum possible entropy and added to loss. We used the most common value **0.01**.

It is important to note that our model took several hours to train to a significant result (30% winrate), therefore the number of our experiments has been quite limited and possibly the choice of some hyperparameters is not optimal, because we noticed a significant variance between different trainings of the same model, and taming this variance would have required several different trainings for each experiment.

Especially the initial part of the training, despite weight initialization, was affected by variance, and luck played an important role in determining the initial learning speed.

The model submitted has been training for approximately 600 iterations of 200 episodes each, which took 24 hours.

3.4 Other Approaches Explored

The first approach explored has been the employment of a simple algorithm based on **Deep-Q Learning (DQN)**, made by 2 fully-connected layers.

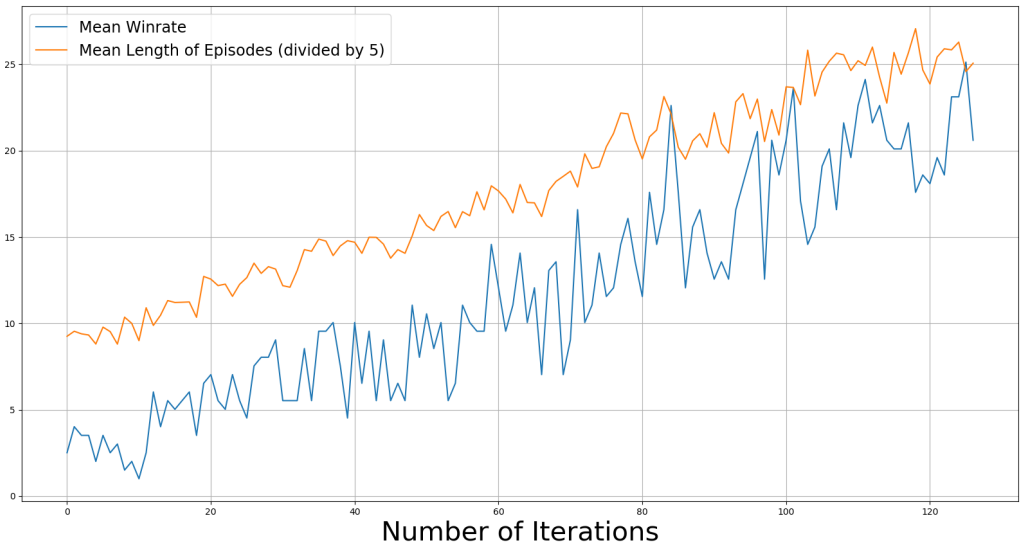
This network received as input the frames **preprocessed through a Convolutional Neural Network**, which was specifically trained through supervised learning to extract the positions of the paddles and the ball. The training data has been extracted from many games played by two slightly different versions of SimpleAI, and this method achieved a precision of 98% in extracting the correct positions from each frame.

This approach was abandoned because the agent was **getting stuck in a local minimum** by always moving up or down. In order to solve this problem we moved to more complex reinforcement learning techniques, and started using directly the frame as input for the network.

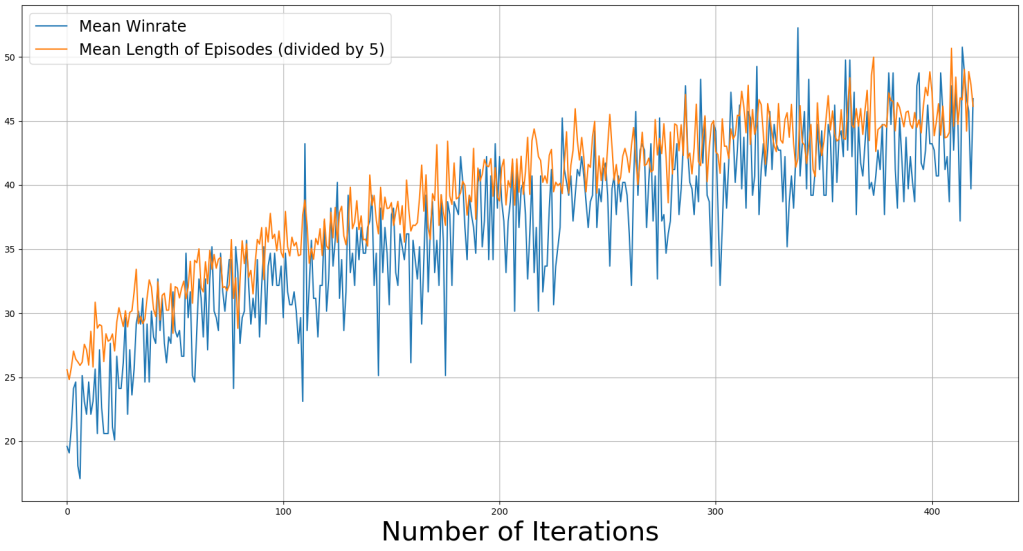
4. Results and Performance Analysis

4.1 Submitted Model Results

The learning curve of the model we submitted can be seen in figures 1a and 1b. The total training time of the submitted model has been approximately 24 hours, and it finally achieved a winrate of approximately 50% against SimpleAI.



(a) Initial part of training of the agent submitted.



(b) Final part of training of the agent submitted.

Figure 1: Learning Curve of Submitted Model

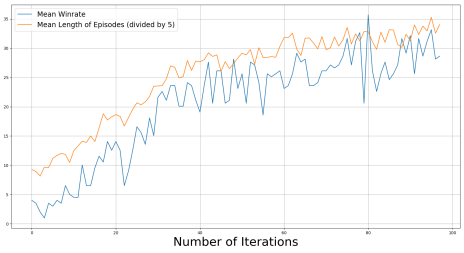
4.2 Hyperparameters Tuning Results

We tried training our model with different hyperparameters to see which version would achieve better results, according to table 2. The model submitted has been the one containing the parameters of line number 4, which was slightly better than the other models, obtaining a winrate of approximately 50%.

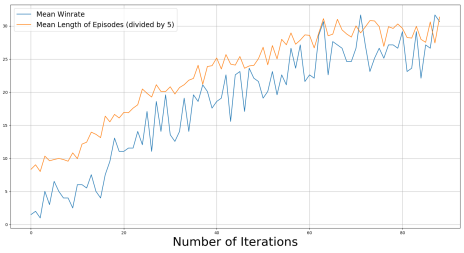
hyperparameters	clipping range	discount Gamma	value coefficient	entropy coefficient	learning rate	horizon	mini-batch	epoch	Preprocessing	policy net architecture	weight init
common range	0.1, 0.2, 0.3	0.99 (0.8 - 0.9997)	0.5, 1	0 - 0.01	0.003 - 5e-6	32 - 5000	4 - 4096	3 -30			
Trainings											
1	0.2	0.99	0.5	0.01	1e-3	200	140 (0.7)	5	2 downsample	512 hid fc	no
2	0.2	0.99	0.5	0.01	1e-3	200	140 (0.7)	5	2 downsample	512 hid fc	yes
3	0.1	0.99	0.5	0.01	3e-4	200	60 (0.3)	12	2 downsample	512 hid fc	yes
4	0.1	0.99	0.5	0.01	3e-4	200	80 (0.4)	10	3 downsample	512 hid fc	yes
5	0.1	0.97	0.5	0.01	3e-4	200	60 (0.3)	12	2 downsample	1024 hid fc	yes
6	0.1	0.99	0.5	0.01	1e-3	200	60 (0.3)	10	3 downsample	1024 hid fc	yes

Figure 2: Table summarizing our hyperparameters tests

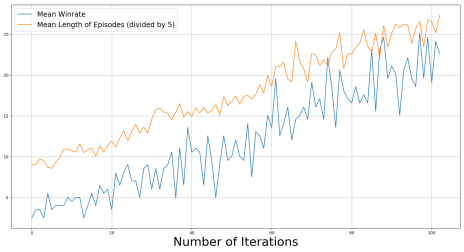
Figure 3 shows the learning curves for the first 100 iterations of the agents with the parameters set according to the table in figure 2.



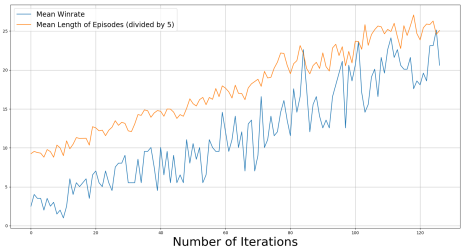
(a) 1.



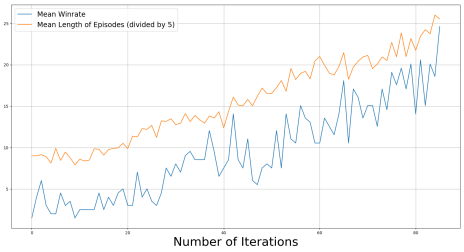
(b) 2.



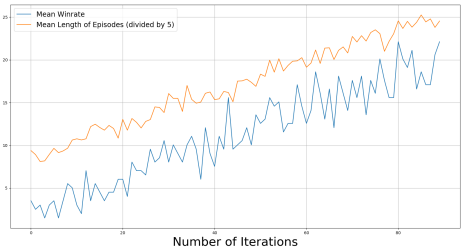
(c) 3.



(d) 4.



(e) 5.



(f) 6.

Figure 3: Initial Learning curve of Hyperparameters Tuning Experiments.

It is easily noticeable that the models with a bigger batch size are more sample-efficient,

anyway those models generated some memory (CUDA) errors after 100-120 iterations, therefore we used the models with lower batch size, because they were slower, but they had the same potential in the long run.

All the trainings we run could have potentially achieved better results if trained for more than one day, or if trained with more efficient techniques like generating experience with multiple agent in parallel, however we didn't have enough time to implement these techniques before the deadline.

As an example, figure 4 shows how one of our trainings achieved more than 5% increase of its winrate after an additional day of training.

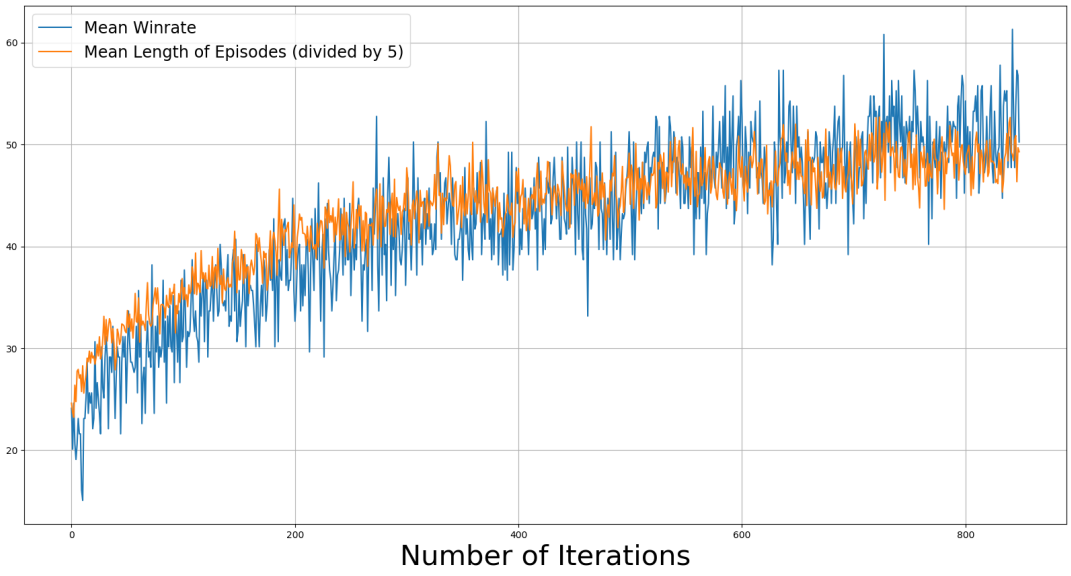


Figure 4: Training after two days (parameters according to line 6 of 2).

4.3 Training Against different models

We tried to resume the training of our models one against each other after having achieved a 30% winrate against SimpleAI (**Adversarial Learning**), and the result can be seen in figure 5. From the result and the rendering, we noticed that our agents rarely manage to play against each other, because *while training against SimpleAI they learned to base their movements on the movements of SimpleAI, rather than on the position of the ball*, therefore when playing against each other, they imitate each other, and just oscillate in the middle of the playing field without following the ball at all.

4.4 Future developments

Solving the problem of the agent learning to follow the movements of SimpleAI, would be an important future development. There are two methods for achieving this result:

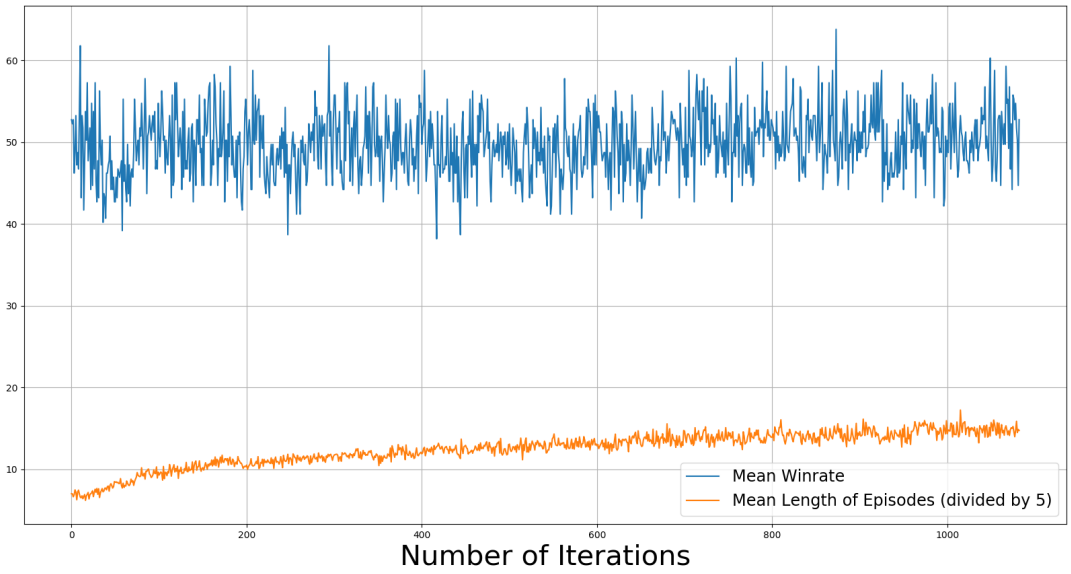


Figure 5: Learning curve of resumed training of two agents with 30% winrate against SimpleAI.

- Training both agents starting from scratch. This has been done by us for one day of training as shown in figure 6, however, it seems this learning process takes a much longer time to achieve good results.

- Another technique we haven't tested is to train the agent by feeding it the cropped frames without the pixels containing the opponent's paddle (rightmost pixels).

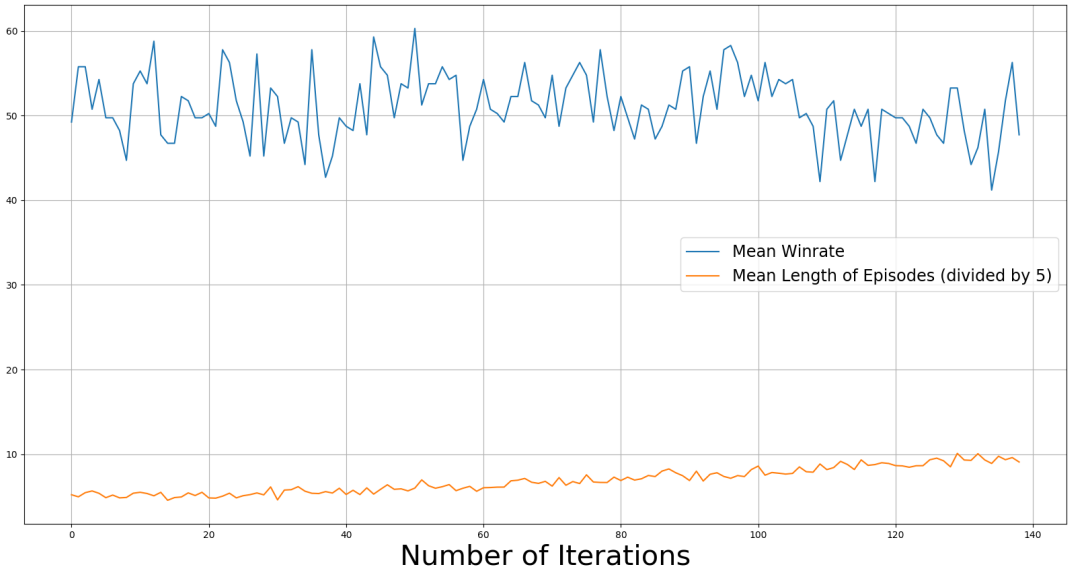


Figure 6: Learning curve of one of two agents learning by playing against each other.

5. Conclusion

Our agent has learned to play pong successfully based on the RL algorithm Proximal Policy Optimization (PPO) in Actor-Critic style, achieving a winrate of 50% against SimpleAI. Since our agent learned to follow the movements of its opponent instead of the ball position, it cannot be better than its opponent. Despite this, we have explored and understood many different RL techniques, which can be generally applied to a number of applications limited only by imagination.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” CoRR, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” CoRR, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.0634

[3] PPO Hyperparameters and Ranges (Medium), Jul 2018. [Online]. Available: https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe

[4] Pong-PPO (GitHub), April 2019. [Online]. Available: https://gist.github.com/s-gv