Reinforcement Learning Project - Two Player Wimblepong

BRENO ABERLE RAÚL AZNAR ÁLVAREZ
breno.aberle | raul.aznaralvarez @aalto.fi

December 22, 2020

Abstract

"In reinforcement learning as in tennis, talent is important, but training makes the master"
As part of this project work a Reinforcement Learning agent is being implemented to play the game of Pong from pixels. In the given environment, the agent controls one paddle and can take one of three actions: moving up or down, or staying in place. The aim is to train a model that is able to outperform the benchmark AI that is provided by the course lecturers and to be a serious contender in the wimblepong course competition playing against other teams.

Contents

1 Introduction

This report covers the scope of the final project in Reinforcement Learning (ELEC-E8125) at Aalto University. Where an implementation of a reinforcement learning algorithm needs to be implemented for an agent to learn how to play the Atari game Pong from pixels. The agent only controls the movement of the paddle, and can take three discrete actions: Up, Down or Stay.

1.1 Motivation

As animals or humans do, in this specific project, the aim is to learn from a visual input, an image captured from an environment, will serve as a basis to understand what is happening around us.

Here, we train to tackle the Pong environment, how to beat the programmed artificial intelligence, by only using images collected from experiences. In other words, how to learn from pixels.

By using this architecture, the problem is scalable not only to the pong interface, but to most of the Atari games and many different environments.

1.2 Problem statement

Preprocessing the frames to obtain information, and train models based on that is a state-of-art methodology in reinforcement learning, and coping with this problem is a key for success for further applications.

To succeed in this task, a proper architecture to handle the different frames need to be implemented. In addition, the frames are used as a basis for an RL algorithm to learn some specific behaviour, in this specific project to beat the provided AI. This adds an extra difficulty since we are not allowed to train information from the environment such as the ball position, which the AI does. To beat the AI, by using Reinforcement Learning we will need to train a model by making him play an extremely high number of games, so that it can learn how to achieve good results.

From a pool of reinforcement learning algorithms, such as DQN, A2C or PPO, identifying which one can deliver the best results in a limited amount of time, is the main issue to address. Some algorithms require an extra effort for tuning their hyperparameters in order to deliver good results, whereas others provide more reliable results, but are less data efficient. Along these report different methods are explained that have been tried out, and their behaviour in the Pong environment.

Once the agent is able to beat the provided AI, a further step is to making it more flexible and being able to compete against different agents with an unseen behaviour. Models tend to overfit if they are only trained against one opponent. Therefore, the trade-off between beating the AI or beating other agents will be taken into consideration along the project.

2 Review of external sources you used

2.1 Convolutional Neural Network (CNN)

The paper [1] introduces the concept of passing observations to a CNN and using the output for updating the policy of the model and getting the action for a specific observation. CNNs are able to retrieve information from data with strong spatial and temporal dependencies. Therefore, this also applies to sequence of images. A CNN can synthesize a complex decision surface like images and can classify high-dimensional patterns [2]. In other words, for data with strong spatial or temporal dependencies, CNNs can output high-level features with important and significant information and reduced data size.

2.2 Deep Q-Network (DQN)

DQN is a combination of Q-Learning with deep neural network architectures [3, p. 437]. Using a convolutional neural network, trained with a variant of Q-learning, has been one of the first approaches to play Atari games with Reinforcement Learning in 2013. DQN achieved results that outperform human

level on many of the Atari games [4]. As a result, the Deep Q-Learning algorithm outperforms all previous approaches on six of the games and surpasses a human expert on three tested games.

2.3 Actor-Critic (A2C)

Actor critic methods are a type of policy gradient methods, that use the estimation of the value function as a baseline to reduce the variance by inducing small bias. It consists of two parts: Critic and Actor. Critic estimates the value function for an specific state-action. Whereas the Actor updates the policy according to the data received from the Critic. To get a graphical and a clear picture about it the following article [5] gives a clear explanation about the process behind the algorithm which makes it more comprehensible , using an analogy of a Fox exploring some unknown environment and how the Actor and the Critic play a roll in the process.

For a more technical approach, the concept of actor-critic methods is as well explained in the book [3, p. 331], where a detailed explanation of the pseudo-algorithm is provided.

2.4 Proximal Policy Optimization (PPO)

PPO is a policy gradient method and optimizes applying a surrogate objective function using stochastic gradient ascent [1]. Policy gradient methods usually perform one gradient update per data sample. However, PPO uses multiple epochs of mini-batch updates. In the application of Atari games it is common practice to collect transitions of a whole game and then applying the policy update. In Pong one game ends as soon one player reaches 21 points. PPO uses the benefits of trust region policy optimization (TRPO [6]), while being simpler to implement and more general. The paper [1] introduces PPO with playing the Atari games and it outperforms other online policy gradient methods. PPO shows beneficial balance between complexity and simplicity. The best scores have been achieved with clipping epsilon of $\varepsilon = 0.2$ and beta was initialized with 1. In the Atari game domain PPO was able to compete in the same level of ACER[7] with similar winning rates.

3 Design of the agent architecture

3.1 Proximal Policy Optimization (PPO)

In the following the most important architecture to realize a PPO implementation are explained. The forward() function receives a state as input and the purpose of this function is to output the action distribution and state value for the input state. First of all, the input state gets passed through the CNN. After that the output gets handed over to the Neural Network layer of the Critic part to calculate the state value. Furthermore, the output of the CNN gets also passed to the Actor Neural Network layer to calculate the action mean. That action mean gets passed to the Categorical class of PyTorch to calculate the action distribution. The action distribution chosen in in this project is the following:

```
1 action_dist = Categorical(logits=action_mean)
```

”Logits” is the log-odds of sampling while ”probs” is the probability of sampling. Based on a discussion with TA’s using probs in combination with with F.softmax() instead of logits is also valid and even common practice. However, in this case logits worked better and was therefore used.

The preprocessing function receives the raw observation image. The input dimensions are 200x200x3. That means the frame has a height and width of 200 pixels and 3 three layers for each RGB color. As first step the image gets gray-scaled and as a result the image only contains one layer anymore with a size of 200x200. As a next step, the image gets resized to the dimension of 80x80 to reduce data size. To further reduce complexity, the the pixel values get converted to either black or white by a threshold. By plotting the processed picture it can be seen that the background is in black and paddles as well as the ball are white.

Moreover, the pixel values get normalized by dividing the pixels by 255. That results in values between 0 and 1.

For training and getting the action stacked images are used to get information about motion, more specifically speed and acceleration. First of all, the new observation is the input and will be preprocessed. Then, the preprocessed image gets appended to the Python data type *deque* of size four, called img-collection. Deque makes sure that the size stays always the same. When appending a new image an entry gets popped with the FIFO principle. That ensures that always the last four images are used. After an episode finishes, img-collection will be emptied. Hence, the function stack-image checks if img-collection is empty. If that is the case, we are at the beginning of an episode and insert four times the first observation to fill up deque with four times the same entry. It is important that there are always four entries, otherwise an error will raise.

In the training.py file the observation, previous observation, action, action probabilities, and binary terminal state attribute get stored in arrays within the agent object. When the update-policy function gets called, the raw images from observation and previous observation need to get stacked. However, by calling the stack-image function within update-policy the attribute self.img-collection-update is used to don't mess up the deque used for training which is self.img-collection. Afterwards, mini-batches are drawn for 10 epochs for the policy update. Each mini-batch draws randomly 40 percent from the stored transitions. Slightly deviation from 40 percent is also valid and shouldn't have a big impact. For each mini-batch, the action distribution and predicted state value for states and next states are calculated with the forward function. By passing the stored action to the calculated action distribution the new action probability is calculated. The advantage is reward plus discounted predicted next state value minus predicted state value

```
# Compute advantage estimates
advantage = old_rewards + self.gamma * pred_next_states_value -
   pred_states_value
```

and the critic loss is the MSE of predicted states value and advantage plus predicted states value:

```
# Critic Loss:
critic_loss = F.mse_loss(pred_states_value, old_rewards+self.gamma*
   pred_next_states_value.detach())
```

Very characteristic for PPO is the clipped surrogate which the following code excerpt illustrates:

```
def clipped_surrogate(self, old_action_probs, new_action_probs,
   advantage):
    # Calculate ratio of new and old action_probs, using exponential, as
        we use log probs
    ratio = torch.exp(new_action_probs - old_action_probs)
    # Clamp ratio, to the desired ratio, 0,2
    clip = torch.clamp(ratio, 1-self.eps_clip, 1+self.eps_clip)
    # Clipped surrogate is minima
    clipped_surrogate = torch.min(ratio*advantage, clip*advantage)
    # Calculate the estimation by taking the mean of all three parts
    loss_ppo = torch.mean(clipped_surrogate)
    return loss_ppo
```

Since in this implementation logarithm is used for probabilities, it is important to take the exponential of log-probs to get non-transformed probabilities. Otherwise, the ratio of the probabilities are different and consequently PPO won't work. This problem got also faced as part of this project. After the ratio is calculated it get clipped between 0.8 and 1.2. The espilon clip value of 0.2 is used in the PPO paper [1]. The clipped surrogate is the minimum of ratio * advantage and clip * advantage. The PPO loss is the mean of the clipped surrogate. Additionally, the entropy of the action distribution is going to be calculated. The whole loss term consists of the PPO loss, critic loss, and entropy loss.

Within the agent architecture the reset() function is used to empty img-collection. The load-model() function allows to load an already trained model. This is necessary to start a training process that continues on the already trained parameters. Moreover, to test a model the agent needs to load a model as well. The get-name() function is mainly used for the wimblepong competition to identify the agent and display its score on the score board. At first, the stack frame function has been called inside the training.py file. However, that doesn't meet the project requirements, because for testing the agent class needs to be able to access all necessary functions. Therefore, all functionalities have been transferred into the agent class.

3.2 Image Preprocessing

In section 3.1 the preprocessing function is explained. After normalizing the gray-scaled image, the pixel contain values between 0 and 1. The algorithm works fine on that. However, to reduce data the values get transformed to 0 or 1 by a threshold. Moreover, the size of the image is reduced by factor 2.5 from 200x200 to 80x80. The reason is to reduce the data size that gets passed to the CNN. That reduces computation and should speed up training. However, other scaling factor around 2.5 can also be used without significant differences in the overall output. To stack four images, different approaches were considered as part of this project. The first approach is to stack frames number 1 to 4, then 5 to 9, and so on. However, that is not good because connection between the consecutive stacks gets lost. The second approach is to stack frames number 1 to 4, then 4 to 8, 8 to 11, and so on to keep the connection. Basically the last image of image i is the first image of stack i+1. However, it is more effort to implement because two environment steps need to be skipped and anyways the provided wimblepong environment has an frame skip of 4 already implemented. Therefore a deque data type is used to pop out the image with the FIFO principle.

3.3 Convolutional Neural Network (CNN)

The preprocessed frames get passed to a CNN. Three convolutional layers and one linear output layer are used to extract features. Hence, the dimension gets lower and the most important features are represented in less data. Consequently, computation and learning will be much faster. For each layer the ReLU activation function is used. The input to the CNN are 80x80 (6400 pixels) preprocessed frames, that are going to be reduced to a size of 512 output nodes. Within the forward function, the output of the CNN is passed to the neural network of the actor as well as to the neural network of the critic part. Three layers are used to extract more information out of the input. The stride is lowered after each layer. The stride determines how many pixels are jumped over to apply the next filter at. Since from layer to layer the size gets reduced, the stride also decreases. After the three convolutional layers a flattening layer is applied to create an output of 512. These numbers can also vary in a small range without having a significant impact. It is important to check the image after preprocessing and before passing into the CNN since it has been a common error source for many teams by experience. For instance, by printing or rendering the image possible errors can be spotted.

3.4 Parallel Proximal Policy Optimization (PPO)

In addition, to the PPO architecture which is already mentioned above, we implemented parallel training, to reduce the correlation. A different training file needed to be used to process, N number of different parallel environments in which our agent interacts with.

Moreover, the agent file also needs to take special care on how the pictures are dealt. Since N number of frames are received for every timestep. Therefore one environment can be done, before another one. Consequently, we modify how the images were processed when the policy gets updated. In simple PPO, the frames stored are in order and is more easy to deal with them. However, for parallel, the frames stored are from different environments and special care needed to be taken in order to guarantee a proper stacking.

We created a global variable list that contained N number of lists, which are the different environments running in parallel. Therefore, when one environment is done, it resets only the list for that environment,

so that the next state for the environment has to initialize the list with 4 new frames. This however, doesn't affect any of the other environments as it is an independent list.

Attached a snippet of the code that tackles this issue, going through the set of raw images and classifying them in their correspondent list.

```
while p<(len(states_raw)):
    for h in range (N):
        state_stacked = self.stack_images(states_raw[p], h, update=True,
            nextstate=False)
        states.append( torch.from_numpy(state_stacked).float() )
        next_state_stacked = self.stack_images(next_states_raw[p], h,
           update=True, nextstate=True)
        next_states.append( torch.from_numpy(next_state_stacked).float()
            )
        if done[p] == 1:  # important to handle episode endings, so that
            for the next one it has to fill with new images
            self.img_collection_update[h] = []
        p=p+1
```

A similar process is carried through the training loop, where when an environment receives the signal that is done, it resets the list only for that environment. As we decided to keep two different global variables to deal with the update policy and the real-time situation.

```
def reset(self, i):
    """ Resets the image collection for the environment which has
       finished"""
    self.img_collection[i] = []
```

3.5 Hyperparameters Selection

To train the model, the selection of the hyperparameters is of high importance in order to achieve results, therefore by looking at the paper [8], and with the advice of the TA's, we ended up using this hyperparameters for the final training:

1. Clipping Surrogate $\varepsilon = 0.2$
2. Epochs Number=10
3. Timesteps Update=500
4. Entropy Coefficient=0.01
5. Critic Loss Coefficient=0.4
6. Parallel Process=30
7. Batch Size=35% of Transition Buffer

3.6 Other architectures used (DQN & A2C)

At the beginning we tried using DQN and A2C architectures. We wanted to start with A2C because the transition to PPO would become natural if we would start with A2C, since not many changes need to be applied. On the other hand, we read and we thought that DQN could probably give some decent results in a quick way, so that we could have something prepared in case we wouldn't succeed with PPO. With the available resources from the exercises and only a bit of tuning of the hyperparamenters we decided to try

both methodologies.

With A2C after most of the trainings the paddle was staying at the middle, since it was assuming that as the best action. It was many times getting stuck in local minima that made impossible to progress.

Although we got some small results with DQN, it was not close to a decent performance and we decided to focus our efforts on PPO. To mention some of the hyperparameters used:

1. Target Update=15
2. Batch Size=128
3. Replay Buffer=70.000

In addition, the ε and the GLIE associated with it were chosen depending on the complexity. If the training was against a Simple AI remaining static a smaller GLIE was chosen since it didn't require that many exploration. But when the difficulty increases more episodes were required, and the GLIE needed to be higher as well to ensure exploration.

4 Training methodology

To train the models, several approaches have been followed through the process. From tweaking the AI to make it more erratic, to make the environment toss the ball to our agent. Starting from scratch against the standard AI, was as well utilized once the architecture of the models seemed to be robust. In addition, as you will see in the subsection 4.4, the utilization of different behaviour of AI's used in parallel environments, which resulted in our final model.

4.1 DQN

For DQN architecture after failing to train it with the standard AI conditions, we decided to opt for the alternative of tossing the ball to our agent (by modifying the environment wimblepong.py) and having the opponent static, so that it would learn that by passing the ball through the other end the sum of rewards would be high. This was achieved as it can be seen in figure 6.

After that, we used that model to try to compete against a weak AI, with a similar parameter to the ball prediction error proposed in the file, we decreased the intelligence of the AI, and once again we served the ball to our agent so that it keep learning the first steps. The performance of this training can be seen in figure 7.

However, once we tried to make the environment toss the ball for both sides our agent got really bad results, and training it from one of the available models was impossible. As is showed in figure 8.

We believe that the training in DQN is tricky with the conditions given, since the experience buffer is lost after a training is finished, the selection of the new ε is difficult to decide, since it may produce to erase our already learnt patterns, but without it we cannot explore new possibilities. Therefore improving our behaviour against weaker agents seems more challenging that with posterior described methods since for policy gradient methods, the policy is stored, and no experience replay is required for their training.

4.2 A2C

We implemented Actor-Critic as a transition to PPO, we didn't have the intention to use it for our final model, but we consider it would be interesting to try to see the results to be able to compare, since it uses the same structure as PPO.

To train it we follow similar procedures with DQN. We trained the model from scratch without modifying the SimpleAi, after more than 250.000 episodes the agent did not learn anything useful, the

episode duration was also invariable. We understood that methodology was like buying a lottery ticket. This can be seen in figure 10.

Consequently, we decided to do the same as with DQN and force the environment to start tossing the ball to us, and playing against an static opponent, an extremely weak opponent, or different variations from them. The end-result, displayed in figure 9, seemed to be similar, not useful behaviour learnt and got stuck in some local minima. The most common one was to stay in the middle.

The last approach that we tried is to make the SimpleAI play some balls as a normal AI, but stop the good behaviour after 'N' number of timesteps, so that it could learn a policy for a bit longer. But it didn't report any positive results.

One important thing to mention, is that along these period we realized as well, that the update of the policy was done too often in our code, due to the difference naming of 'episode' in external sources of Atari Pong. Therefore we changed the update to 500 timesteps, which would then be more accurate.

After seeing no progress, we decided to not spend more time on train with A2C and focusing to a more robust method as PPO.

4.3 PPO

Since our main goal was to focus on a robust method like PPO, we decided to move to it as soon as no results were given in any of the above methods mentioned.

To train with PPO, we followed the same steps as with the other methodologies. By starting with tossing the ball to our agent and playing against a static opponent. After the correction of some mistakes in our implementations (regarding the logarithmic probabilities), the results were really positive, as can be seen in figure 11 .

Next step, we used that model and trained it on top against a weaker AI with a high ball prediction error. We used both methodologies of serving both sides, or only to us, but it didn't seem to make a big difference. The results of the training serving both sides against a weaker AI can be seen in figure 12. Special attention should be paid to the timesteps plot as well, where it can be seen how the duration of the points get longer.

Last, we trained using the previous model against the standard provided SimpleAI, the result which is shown in figure 13, provided us with a winning rate slightly around 50%.

We tried to train on top of that model and got to improve our performance but not further than 55%. Thus we decided, to make an extra effort to jump to parallel to break the correlations that single training can generate.

4.4 Parallel PPO

Since simple PPO gets stuck to some learnt behaviour and some correlations learnt through the whole training, we decided to implement PPO parallel as above explained.

By applying the parallel environments the results were improving to an average of 70% against the simple AI, by training it from scratch. Since training it on top of the previous single PPO model didn't improve as much. Figure 14 shows the results.

However, when testing it against himself the result was atrocious, from an average episode of 300-400 timesteps, to not being able to give back the first ball in many occasions.

We try to train against himself, but as its behaviour against another agent was so bad, that was not helpful. And it may have learned to beat some agents, but the behaviour against the simple AI would have dropped considerably.

However, we came up with an interesting idea. Introduce different 'personalities', in the simple AI file, when it initializes it chooses one, therefore by using parallel environments and using different behaviours that could be more similar to how other agents act. For example, wait until the ball comes back to move again to the desired direction, or hit the ball with the edges so that is more unpredictable. Six different behaviours were introduced into the simpleAI file.

Nevertheless, we wanted to make it if something, overfitted towards simple AI, at the end that was the first opponent to beat. So we gave more chances so that, this specific behaviour would be choose from the

pool.

To get the idea of the concept, a small part of the code is displayed below to give the reader a better picture of the idea behind that.

```
"""Distribution of probabilities, so that the behaviour 1 is more
   plausible(Normal AI). The other we give more importance is 3"""
self.personality = random.choice([1, 2, 3, 1, 4, 1, 5, 6, 1, 1, 3])
```

An example of one of the behaviours:

```
if self.personality==5:
    #As the height is 20, we choose the half, so that it tries to hit it
        with the edge
    y_diff = my_y - ball_y+10
    #Checking if the ball is coming to us or going away
    x_diff_before = my_x - self.anterior
    x_diff = my_x - ball_x
#If ball is coming to us, AI will hit it aggressively with the edge,
   simulating agent behaviours
    if abs(y_diff) < 2:
        action = 0  # Stay
    else:
        if y_diff > 0:
            action = self.env.MOVE_UP  # Up
        else:
            action = self.env.MOVE_DOWN  # Down
#If the ball is moving away, then do random actions(mostly staying)
    if abs(x_diff)>abs(x_diff_before) :
        action = random.choice([1, 2, 0 ,0, 0, 0, 0, 0])
```

As we ran 25-30 parallel environments, depending on CPU capabilities, we could introduce the different behaviours in some environments, those would be captured, and the model would be trained at the same time considering all of them.

This gave amazingly good results in the first run, show in figure 15, and after not many episodes. We were expecting to have a trade-off between, performance vs AI and performance vs Agents. But surprisingly the model cope with both extremely good and improved both performances.

This has been the selected methodology to train our final model shown in figure 16, which was trained on top of figure 15. We did not train it from scratch using different behaviours, rather just training it on top of the parallel model against the SimpleAI with standard behaviour.

5 Evaluation of results

The figures 1 and 2 show the average reward and average timesteps duration respectively , for the end model against the simple AI, for a testing run over 1400 episodes.

An average reward around 8.35 was recorded, which translates to a winning rate over 91 %. The episode length settles to approximately 400 timesteps per episode, meaning that the duration of every point is considerably long.

Extra runs have been recorded with similar values, with a variance of 3-5% in the WR, if enough episodes are played.

$$WinningRate > 91\% \tag{1}$$

$$Episode\ Length = 400\ timesteps \tag{2}$$

To simulate what it could be the performance against an external agent, we tried to use an specific behaviour of the AI that reproduces similarly what an agent could perform, trying to hit with the edge and stop in the position after hitting the ball. With that and acknowledging, that do not guarantee those performance against all different agents. The results are displayed in figures 3 and 4. With an average reward of 5, we can derive a winning rate around 75%. Surprisingly we see an increment on the episode length achieving around 600 timesteps average.

$$WinningRate = 75\% \tag{3}$$

$$Episode\ Length = 600\ timesteps \tag{4}$$

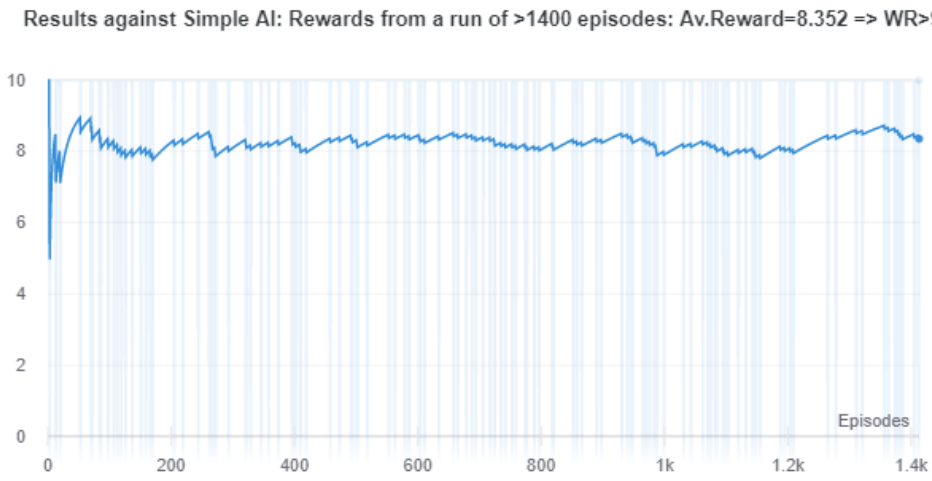


Figure 1: Testing results by using the model trained in 16 against the SimpleAI (Normal Behaviour).

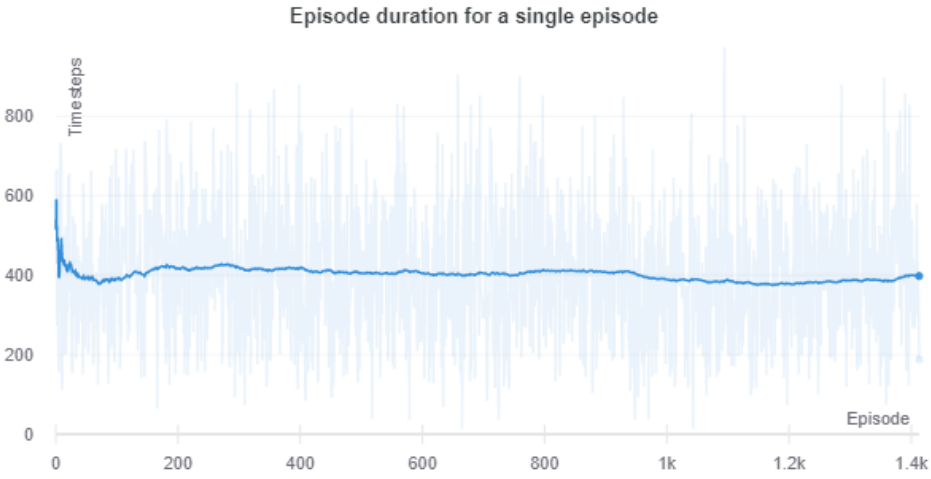


Figure 2: Average timesteps results by using the model trained in 16 against the SimpleAI (Normal Behaviour).

To understand the importance of the behavioural AI's, we show in figure 5 the testing results of the algorithm trained in parallel against the SimpleAi. Although logically this model should be overfitted towards SimpleAI and achieve same or higher results. It did not, since by training with different agents it started to learn as well some tricks that like hitting more often with the edge that made the SimpleAi more defeatable. By training against the SimpleAi, we obtained around 70% of winning rate.

$$WinningRate = 70\% \tag{5}$$

Therefore, the results show how the impact of adding different behaviour affects as well the results. And the game changer that this supposed to our model.

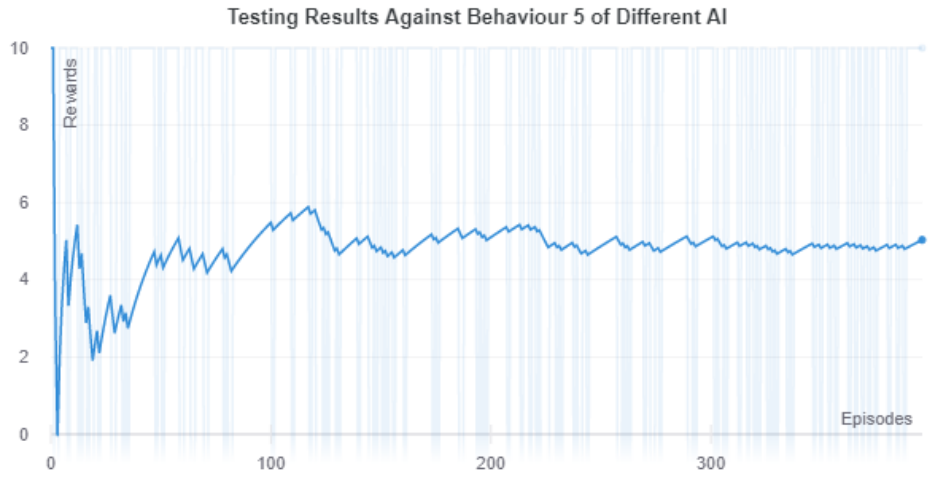


Figure 3: Testing results by using the model trained in 16 against different behaviour more similar to a RL algorithm.

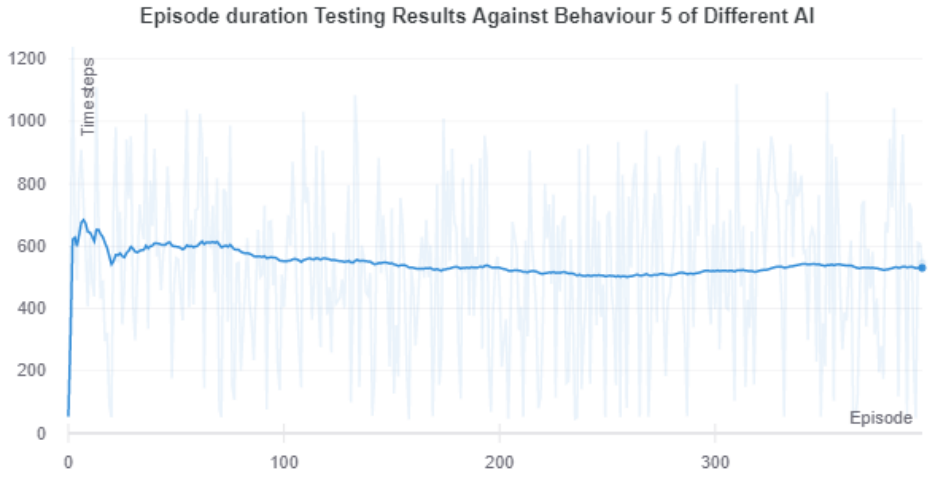


Figure 4: Average timesteps results by using the model trained in 16 against different behaviour more similar to a RL algorithm

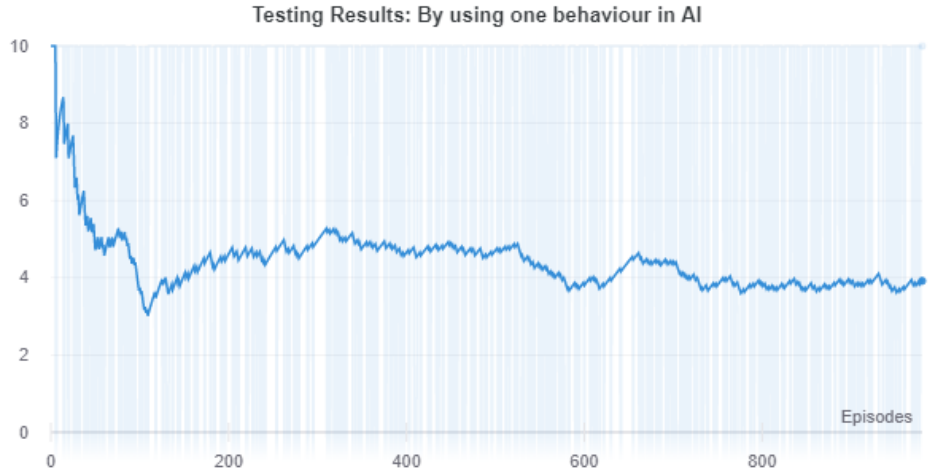


Figure 5: Testing results by using a model trained against SimpleAi

6 Conclusions

We believe that the best approach for this task is to use parallel environment and train against different agents in the same run. This will make the algorithm to generalize better, by only changing the behaviour of the AI and use that file we increased our winning rate considerably against the provided Simple AI, but

as well against other behaviours of the AI or other agents.

The fact that the agent performs so bad against agents if it is only trained against the simpleAI, is due to unvisited states. In other words, by training only against the SimpleAi, it has never experienced that the opponent paddle is in a *y position* different than the *y position* of the ball. This correlation and assumption done in training, needs to be torn apart, if we want our agent to generalize better and perform in a more accurate way.

However, which implementation works the best may depend on many hyperparameters, training capabilities (storage and training time). PPO guarantees results, is simple to implement, and it is widely used for this type of applications such as in DOTA [9]. As by training in parallel with different environments that differ more between them, other algorithms can turn wrong. The clipped function makes the policy not to change in excess compared with ACER[7] for example. Which lead to a higher stability along the process

As will be further explained in the limitations section, the use of DQN[4], DDQN[10] or Rainbow[11], we couldn't consider. Since achieving reliable and good results, especially against other agents seemed difficult with the limitations we had, as the training was interrupted with our available options to run the codes, which lead to the loss of the experience buffer. But with enough computational power, and by using parallel environments, which is not something really popular in DQN methods, it may outperform PPO with a fine tuning in the election of hyperparameters. Especially Rainbow or DDQN that may be more stable and reliable than standard DQN.

To sum up, parallel PPO guarantees results and it is easy to implement compared to other methods, it is not as sample efficient as other policy gradient algorithms such as ACER, which perform really well under Atari environments by using experience replay in actor critic methods. DQN was the first method to perform well in Atari environments, but it tends to overestimate action values under certain conditions, whereas DDQN reduce this over-estimation and provides much better performance. Rainbow, provides really good results but its implementation is way more complex than the previously mentioned, and is a state-of-art method in reinforcement learning.

7 Limitations

The major limitation faced during the project was the training servers used, we connected to one of the Linux computers with ssh(lyta/kosh). And specially for DQN training the fact that the process was killed after 12 hours, independently on how did we set the priority or if the screen command was used, was difficult to cope with. This could be caused due to high traffic in the servers since many project deliveries not only for this course but for others such as Big Data use this resources and then the process ends earlier. For DQN this was leading to us, to lose our experience buffer, as well as not being able to set up a proper GLIE function.

Another limitation that can be seen in the training plots, is the fact that the plots are cut in parts since after 12 hours we were ejected from them. For policy gradient methods is not problematic in terms of behaviour but it gets more difficult to have an overall picture of the training by looking at 3 different plots.

Parallel training offered some limitations when training with the GPU, since it doesn't have enough memory to deal with an enormous amount of frames, whereas the CPU could handle around 20-30 parallel environments.

The last limitation we found was to train against different agents in parallel, this is something we will further investigate. The trick of setting them in one file may be the solution. But the setup also for the parallel environment files need to be changed to take two actions, deliver two rewards, for each of the different environments was too complex for us to handle it appropriately in a reasonable time. Therefore we decided to stick with changing the behaviour of the AI.

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278 – 2324, 12 1998. doi: 10.1109/5.726791

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

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[5] R. Gilman. Intuitive rl: Intro to advantage-actor-critic (a2c). [Online]. Available: https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752

[6] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

[7] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," 2017.

[8] H. Tang, Z. Meng, J. HAO, C. Chen, D. Graves, D. Li, W. Liu, and Y. Yang, "What about taking policy as input of value function: Policy-extended value function approximator," 2020.

[9] B. Chan, "Openai five," Oct 2020. [Online]. Available: https://openai.com/blog/openai-five/

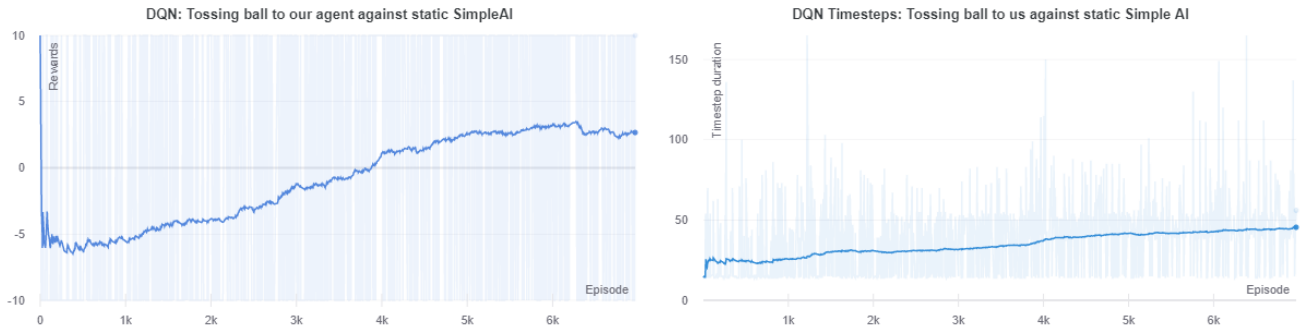
[10] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[11] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," 2017.

A Appendix

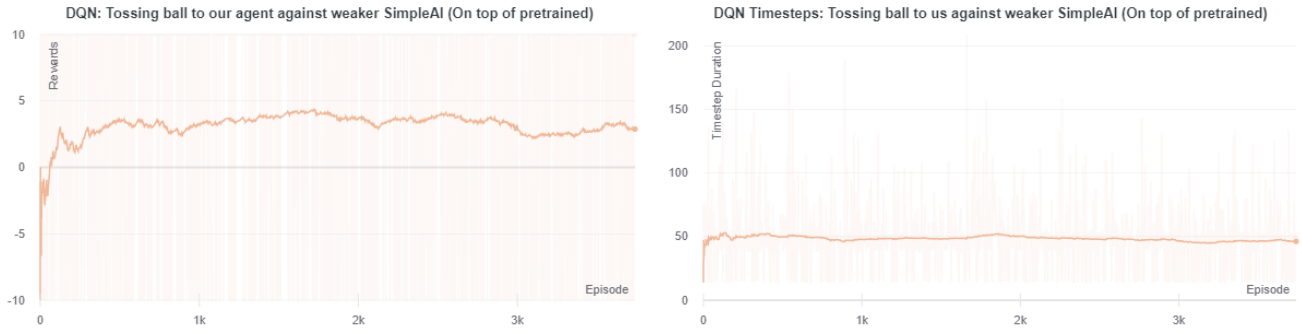
A.1 Plots

A.1.1 Plots DQN



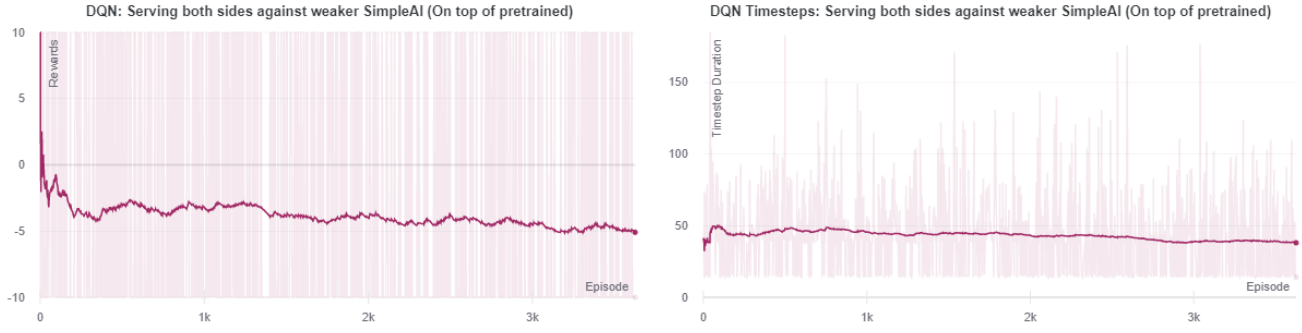
(a) Rewards (b) Timesteps

Figure 6: Training performance plots with DQN in single environment. Training from scratch and environment serving ball always to our agent against static opponent.



(a) Rewards (b) Timesteps

Figure 7: Training performance plots with DQN in single environment. Training on top of pre-trained model of figure 6 and environment serving ball always to our agent against against weaker SimpleAi with bpe = 30 .



(a) Rewards (b) Timesteps

Figure 8: Training performance plots with DQN in single environment. Training on top of pre-trained model of figure 7 and environment serving ball both sides against against weaker SimpleAi with bpe = 30 .

A.1.2 Plots Actor-critic

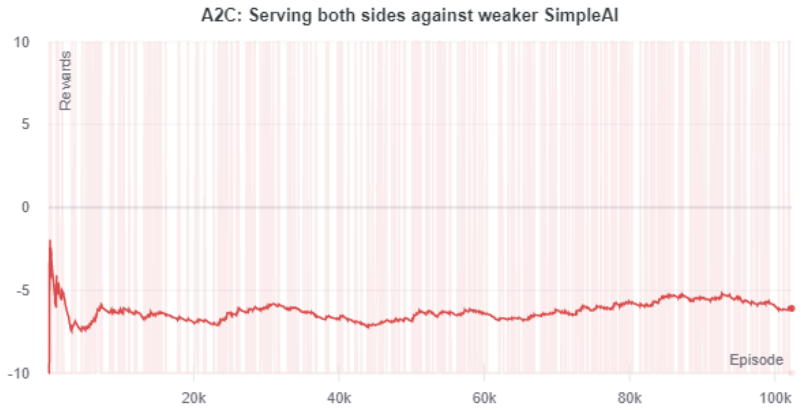


Figure 9: Training performance plots with A2C in single environment. Training from scratch and serving to our agent against weaker SimpleAI with bpe = 50.

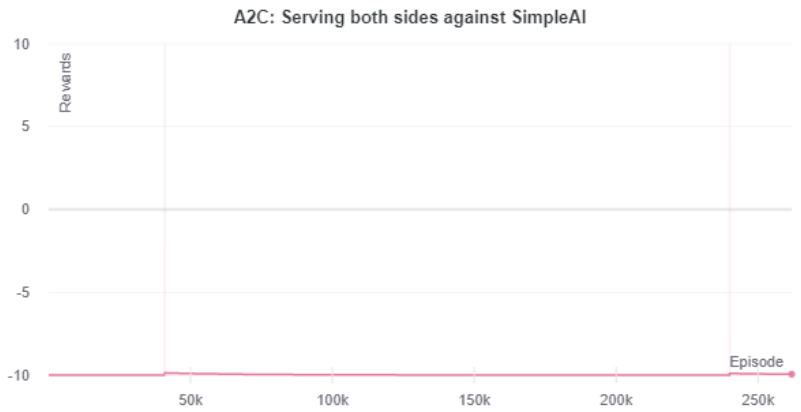
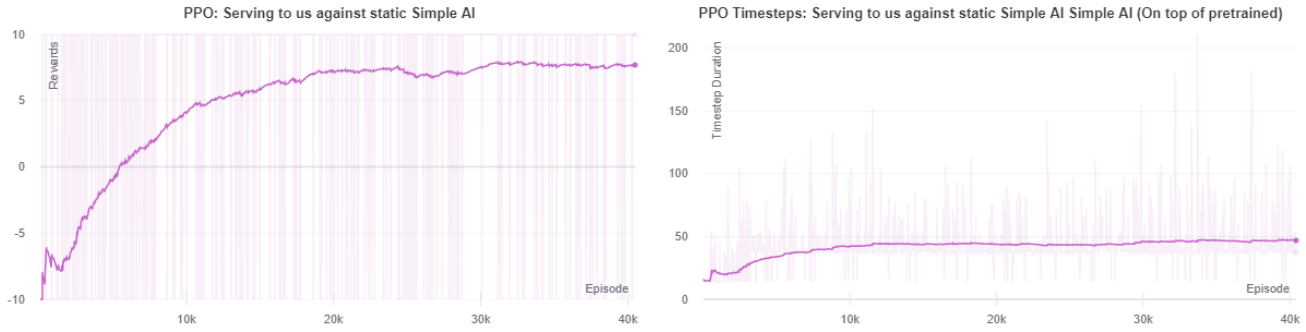


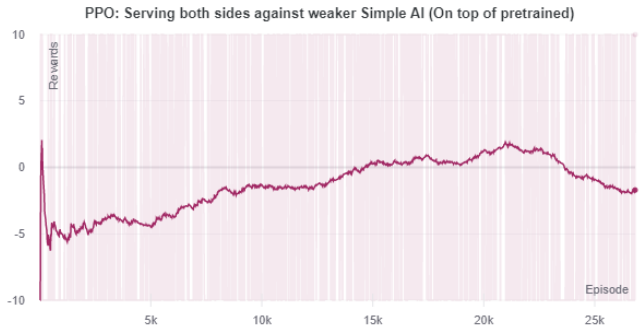
Figure 10: Training performance plots with A2C in single environment. Training from scratch and serving both sides against SimpleAI.

A.1.3 Plots PPO single environment

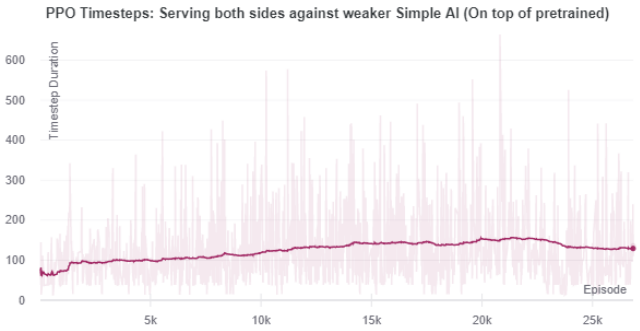


(a) Rewards (b) Timesteps

Figure 11: Training performance plots with PPO in single environment. Training from scratch and environment serving ball always to our agent against static opponent.

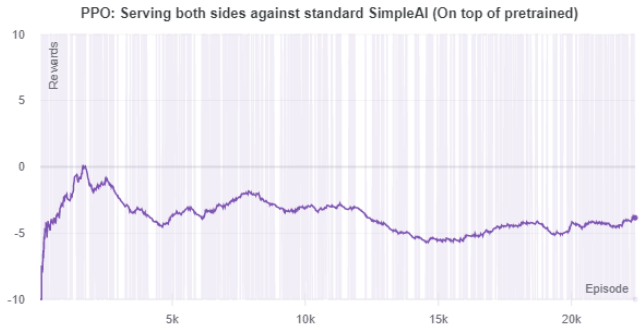


(a) Rewards

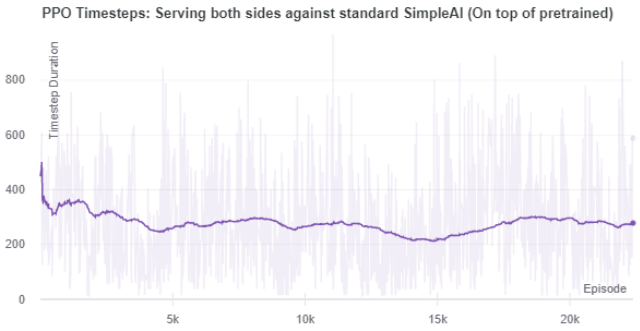


(b) Timesteps

Figure 12: Training performance plots with PPO in single environment. Training on top of pre-trained model of figure 11 and environment serving ball both sides against weaker SimpleAI with bpe= 30.



(a) Rewards



(b) Timesteps

Figure 13: Training performance plots with PPO in single environment. Training on top of pre-trained model of figure 12 and environment serving ball both sides against SimpleAI.

A.1.4 Plots PPO parallel environment

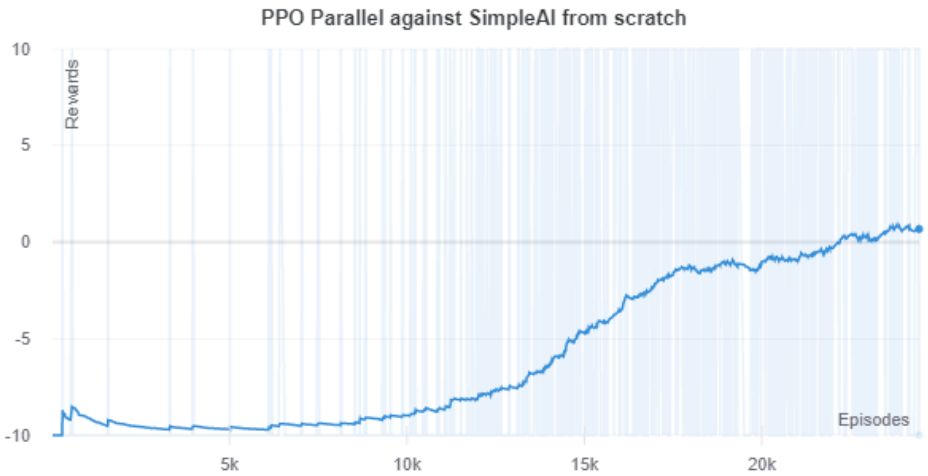


Figure 14: Training performance plots with PPO in parallel environment. Training from scratch against SimpleAI.

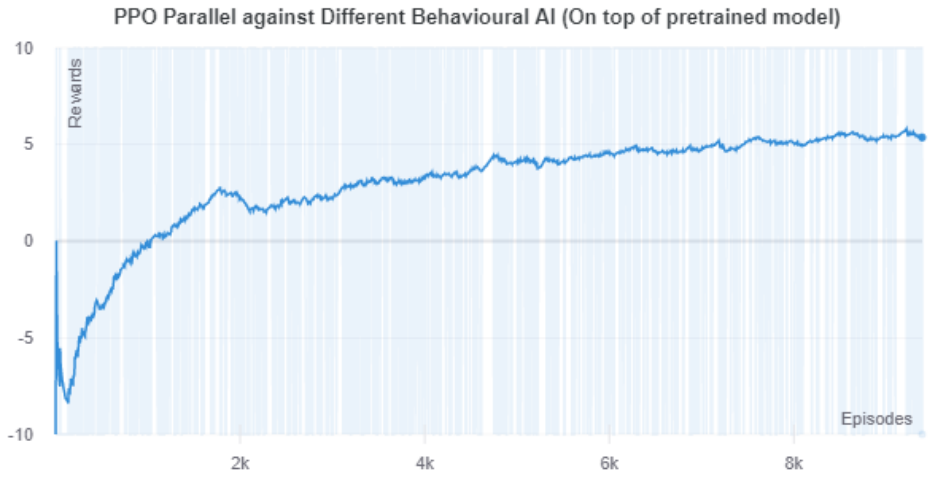


Figure 15: Training performance plots with PPO in parallel environment. Training on top of pre-trained model of figure 14 against different behavioural opponent AIs.

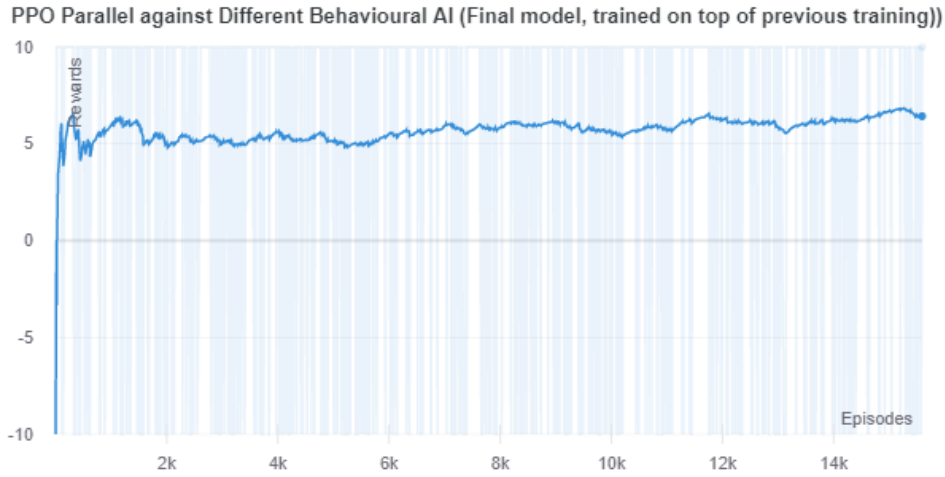


Figure 16: Training performance plots with PPO in parallel environment. Training on top of pre-trained model of figure 15 against different behavioural opponent AIs. Final training.

A.2 Source code

The source code for our Reinforcement Learning project can be accessed through the following GitHub repository: `https://github.com/breno-aberle/rl-pong-project`