Advanced Topics in Computational Intelligence

Reinforcement Learning Algorithm Implementation

Academic year: 2021/2022

Author: Joaquim Marset Alsina

1 Introduction

The present document contains the details, results, and discussion of the first assignment of the *Advanced Concepts in Computational Intelligence* subject. It consists in implementing and testing a deep reinforcement learning algorithm from the first part of the course. We have decided to implement Proximal Policy Optimization (PPO) [3]. To test the results of our implementation, we have used some OpenAI Gym [1], and ViZDoom [2] environments. In section 2 we explain the selected environments. In section 3 we explain the PPO algorithm. Finally, in section 4 we show the results of the algorithm in those environments.

2 Environments

We have decided to test our implementation with environments presenting a variety of state and action spaces. In particular, from OpenAI Gym we have selected LunarLanderContinuous-v2 and BipedalWalker-v3, both having continuous states and actions in the form of a vector. From ViZDoom we have selected "basic" and "health_gathering". All the ViZDoom environments use game frames as states, and the action space is discrete. Therefore, we are not only learning from float vectors but also frames.

We would like to also test it with more complex environments like some Atari games or some MuJoCo environments. We leave it as future work we would like to do.

2.1 LunarLanderContinuous-v2

The environment consists of a lander that must land on a pad on the moon's surface, always located in the same coordinates, but the shape of the surface around may change (useful to avoid memorization and improve generalization). We can see an example of surface in Figure 1

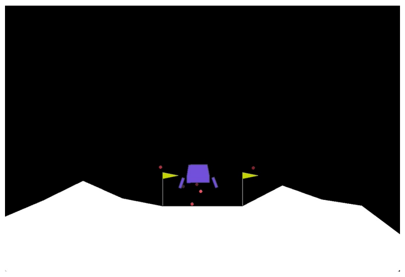


Figure 1: LunarLanderContinuous-v2 environment

The lander has a primary engine in the centre and a secondary engine on each side. It also has two legs that should touch the ground when landing. The episode ends when the lander crashes, steps too far away from the screen, or it lands (on the surface or the landing pad). The yellow flags delimit the landing pad.

The state consists of six real values (taking values from negative to positive infinity) and two binary values. In particular, it contains the lander coordinates, vertical and horizontal speed, angle and angular velocity with respect to the ground, and a sensor for each leg indicating if the leg touches or not the surface.

The action is composed of two real values ranging from -1 to 1. The first one controls the main engine, having a sub-range where the engine is off, and another where the throttle is applied at different levels. The second value controls both side engines, one sub-range for the left engine, another for the right, and a third where both are off.

The environment gives different rewards depending on some conditions. If the lander lands, the environment gives +100 points, and if the lander crashes, -100 points. The fuel is infinite, but we want to discourage the lander from spending too much. That is why the environment gives a negative reward at each step for using the engines, the value being higher with the main engine (it consumes more). We want to land in a particular position, so the environment gives a negative reward for stepping away from the goal with respect to the previous step. Finally, it gives a positive score for each leg touching the ground (a safer landing).

[1]`https://www.gymlibrary.ml/`
[2]`http://vizdoom.cs.put.edu.pl/`

2.2 BipedalWalker-v3

The goal is to train a bipedal robot to walk through a slightly uneven terrain. We show a frame of the environment in Figure 2

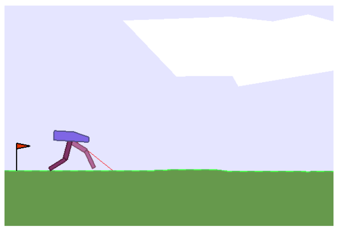


Figure 2: BipedalWalker-v3 environment

Both the state and action space are continuous, and like the previous environment, are represented with vectors of real values. In particular, the state is composed of twenty-four values:

- hull angle speed: ranging from 0 to 2π
- angular velocity: ranging from -inf to +inf
- horizontal and vertical speed: each ranging from -1 to +1
- angular position of the two joints of each leg: each ranging from -inf to +inf
- angular speed of each joint: each ranging from -inf to +inf
- binary value for each leg indicating if it is touching or not the ground
- 10 lidar measurements: each ranging from -inf to +inf

There is no information about the terrain, so the agent does not know where the robot is running, and it needs to use the lidar readings. As mentioned before, the robot has two legs, and each leg has two joints (one in the hip and one in the knee). The action contains four values with the torque to apply to each of the four joints. The torque can take values in the range [-1, 1].

The environment rewards the robot for moving forward, reaching more than 300 points at the end of the course. If the robot falls, the agent receives -100 points and the episode ends. Applying motor torque costs some points, so the idea is to apply the torque when needed to obtain a better score. As expected, the idea is for the robot to walk as far as possible, and try to walk in an optimal way to maximize the score. There are different known ways in which the robot can learn to walk, which can be found in [3].

2.3 ViZDoom environments

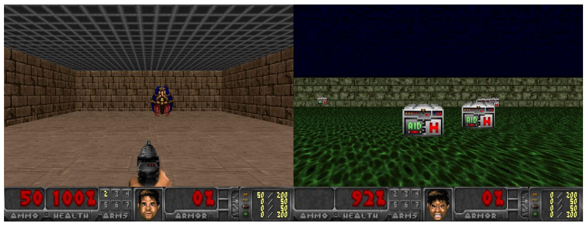
ViZDoom consists of certain mini-games based on the original game DOOM, where we can apply reinforcement learning. In all these environments, the state is a game frame, and the action space is discrete, representing an action as a one-hot vector. Each environment defines which actions are valid. The delivered `.cfg` files contain the definition of each environment.

We have tested *basic*, the easiest environment where we have to kill a static monster always located at the same position. The agent appears in some random location in the room, so it has to move and shoot to kill it. Therefore, the valid actions are moving left or right and shooting the gun. Each time step costs -1 point, shooting costs -5 points, and killing gives +100 points. Therefore, the maximum reward an episode can produce is 95 when the agent is placed in front of the monster and kills the monster in one action. If the agent fails when shooting, it receives -6 points. The ammunition is limited, so if the agent does not kill the monster, the episode will end in more than -300 points.

We have also tested *health_gathering*, where the player is placed on an acid pool, and it progressively loses health. Fortunately, there are multiple health packages restoring the player health points. Each step rewards the agent for being alive with +1 point. Therefore, the agent has to survive as long as possible, being +2100 the maximum reward corresponding to the episode time-steps. The only actions the agent can do is turning left or right, and moving forward.

We show a frame of each environment in Figure 3.

[3] `https://pylessons.com/BipedalWalker-v3-PPO`



(a) basic (b) health_gathering

Figure 3: ViZDoom *basic* and *health_ gathering* frames

3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy actor-critic algorithm developed to simplify the second-order optimization of Trust Region Policy Optimization (TRPO) while retaining data efficiency and reliable performance.

Actor-critic methods directly learn a policy (i.e. a probability distribution for each state) instead of extracting it from the state-value function ($V(s)$) or the state-action value function ($Q(s,a)$). The actor learns the policy, and the critic learns the state-value. Plugging the approximated state-value (or some other value) to the actor loss, the critic guides the learning of the actor (explaining why it is called "critic").

PPO uses the advantage function (i.e. $A(s,a) = Q(s,a) - V(s)$), which determines if taking some action in a particular state is better than taking the expected for that state. If it is better, the probability for that action will be increased and reduced otherwise. Thanks to expressing the advantage as $A(s,a) = r + \gamma V(s') - V(s)$, we do not need to estimate both Q-values and V-values, and we simply learn the V-values.

Both TRPO and PPO try to make the highest possible policy improvement without stepping too far to cause the performance to collapse. As we have said, PPO uses a simpler method that empirically seems to perform at least as well as TRPO in keeping new policies close to the old ones. To do so, the authors of PPO defined two versions in which different actor losses were used. In this assignment, we have implemented the same the people from OpenAI used, which uses the Clipped Surrogate Objective loss. This loss is defined as follows:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

$$L_{CLIP}(\theta) = E(min(r_t(\theta)A_t,\ clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$$

The first equation defines a probability ratio for some action under the current policy and under an older policy. When the action is more probable under the current policy, it will be greater than 1 and vice versa. This function appears as a substitute of the $log\pi_\theta(a_t|s_t)$ that we can find in the vanilla policy-gradient. Therefore, we end up with the loss function $r_t(\theta)A_t$, being A_t the advantage for the current action. TRPO uses this same loss function subject to some constraints that make the method difficult to implement.

The second equation uses the ratio and the advantage to compute the final actor loss. The idea is that in a training step, using a particular batch, we could have some action that is a lot more probable under the current policy (or a lot less probable), taking a big gradient step that could cause the mentioned collapse. To avoid this collapse, we introduce the clipping of the ratio between $1-\epsilon$ and $1+\epsilon$, removing all the TRPO constraints by adding this term directly in the loss. ϵ is a hyper-parameter that the PPO authors find 0.2 to be a good value. The effect of this clipping can be seen in Figure 4:

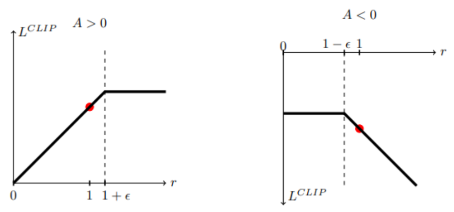


Figure 4: Clipping effect (taken from [3])

When the advantage is positive, we want to make that action more probable. However, we limit how more probable we make it by clipping L_{CLIP} when the ratio exceeds $1 + \epsilon$, causing the derivative to be 0. When it is negative, we want to reduce the action probability. Again, we limit how less probable it becomes by clipping L_{CLIP} when the ratio is under $1 - \epsilon$, generating a 0 derivative. Therefore, these clipping regions prevent being too greedy and update the policy too much at once. If not, we could destroy the good estimation of the policy just for a particular encountered batch.

If we look again at the right plot of Figure 4, we can see how the ratio can infinitely grow even though the advantage is negative. This situation would happen when we increase a lot the probability of an action, worsening the policy. In that case, we would like to undo the gradient step. Luckily, the loss function would be negative there, so the gradient would step in the inverse direction, making the action less probable by the same amount we screwed up. It is the minimum operation in L_{CLIP} the one that undo this mess, as the value will be smaller than the clipped version. We can find the same "undo" region in the left plot. Without the minimum, we would always clip the loss, making the regions flat and not fixing the mistakes.

Thanks to the Clipped Surrogate Objective, PPO can run multiple epochs of gradient descent with the same set of samples without causing destructive policies updates, squeezing more out of the collected samples. Therefore, we will be updating the policy multiple times with the same sample set. However, we can still consider the algorithm on-policy even though we collect the samples with an outdated policy thanks to performing small policy updates.

In Figure 5, we present the algorithm in pseudo-code. As we can observe, PPO runs multiple actors in parallel, each collecting data and then gathering and sampling all that data to perform the updates.

```
**Algorithm 1** PPO, Actor-Critic Style
for iteration=$1, 2, \dots$ do
    for actor=$1, 2, \dots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 5: PPO algorithm in pseudo-code (taken from [3])

3.1 Implementation details

Our implementation is based on the OpenAI Spinning Up implementation [4], but with some modifications and details to consider. In particular:

- As [3] specifies, we use the Generalized Advantage Estimator (GAE) [2] as an estimation of the advantages to avoid looking beyond the limit of transitions we perform in an iteration. However, when sampling a batch we decide to normalize to zero mean and unit variance to ensure better that the algorithm can control the policy updates.
- Instead of running multiple agents to sample transitions, we use multiple copies of the environment (setting different seeds to account for some variation) and a single learning agent. Therefore, the agent acts in parallel in all the environments collecting experiences. Similar to a modification of Asynchronous Advantage Actor-Critic (A3C) [1] to avoid having data synchronization problems.
- To update the critic, the authors used the Mean Squared Bellman Error (MSBE). That is, computing a TD(0) error bootstrapping the target. Instead, we use the loss function the people from OpenAI implemented in their faster GPU-enabled version of PPO, called PPO2 [5]. It uses the same idea of limiting the update by clipping the value using ϵ. Also, they use the discounted long-term return instead of the bootstrapped target. Therefore, they calculate the clipped and un-clipped squared error, select the maximum value between them, and calculate a mean.
- Even though the Clipped Surrogate Objective should be enough to avoid the policy collapse, it can still happen. That is why we follow what the authors of [4] did to try to enforce even more small policy updates. They decided to include some TRPO constraints using the Kullback–Leibler divergence [6] (calling

[4]`https://spinningup.openai.com/en/latest/algorithms/ppo.html`
[5]`https://github.com/openai/baselines/blob/master/baselines/ppo2/model.py#L68`
[6]`https://en.wikipedia.org/wiki/Kullback-Leibler_divergence`

the variation TR-PPO) to stop updating the policy (making the gradient 0) when the divergence is above a certain threshold.

- We apply annealing in the learning rate and the ϵ we use to clip both actor and critic. We have computed it as the factor $1 - i/I$ being i the current iteration and I the total number of iterations. This factor is then multiplied by the original learning rate and ϵ to get the annealed value.

4 Experiments and Results

Before diving into explaining and discussing the results of each environment, let's mention some common aspects.

As we saw in class, we need to ensure the algorithm works by training the agent multiple times with different seeds. For this reason, we repeat the training process three times, setting random seeds to each environment each time (avoiding, if possible, the agent memorising it). Also, the networks are initialized random, so we also ensure variability there.

When reporting the agent performance, we will focus on the evolution of the average reward of the last hundred episodes. However, we will also include the actor and critic loss evolution and the KL-Divergence. As we repeat the process three times, we will plot them as the mean $\pm$ the standard deviation. But the reward will be plotted on an episode basis, whereas the other three on an iteration basis. We run the algorithm for some iterations and train the model at the end of each iteration. Therefore, an iteration will include multiple episodes.

Finally, each training repetition will stop when we reach a maximum number of episodes. Even though the algorithm runs for several iterations, each training process can learn at a different rate, needing more or fewer iterations. Therefore, to compare the results, we will set the needed number of iterations to reach the desired number of episodes.

4.1 LunarLanderContinuous-v2

In this environment, we ran two experiments until solving the environment. This environment is solved when the agent achieves an average of 200 points over 100 consecutive episodes.

In the first experiment, the results were really bad, not being able to stabilize, and in fact, the performance crumbled. We were not re-scaling the rewards, which has resulted in the critical point in solving it. Re-scaling the reward and the gradient clipping allowed us to solve it.

In Table 1 we show the parameters we set in each experiment, each running for 5000 episodes. Regarding the other parameters, we use 8 parallel environments, set the learning rate to 1e-4, the discount factor γ to 0.99, and the GAE λ to 0.95. These last parameters are always the same for all the environments, so we will not explain them more.

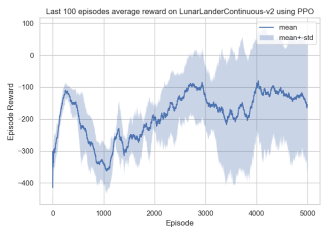
Parameter	Experiment 1	Experiment 2
Reward scale	1	0.01
Buffer size	512	512
Batch size	512	512
Gradient clipping	0.5	50.0
Max KL-Divergence	0.15	0.15

Table 1: LunarLanderContinuous-v2: Experiments hyper-parameters

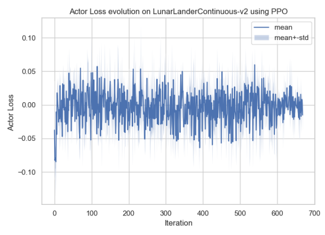
We define the buffer size for one parallel environment, so the total number of collected experiences is multiplied by 8. The batch size is 512, which is indeed high. However, we previously worked with this algorithm, and we saw overall decent results. Also, having to process fewer batches accelerates the training process.

The network architecture is the same for both actor and critic, and it is mainly composed of two dense layers with 256 and 128 units. Each layer uses LeakyReLU non-linearity and batch normalization. Then we have two dense layers with `tanh` activation to generate the Gaussian mean and logarithm of the standard deviation. BipedalWalker-v3 shares the same architecture, so we do not explain it there.

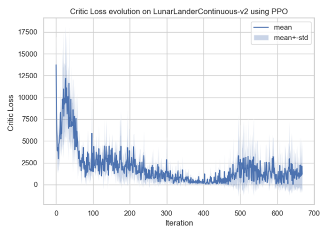
In Figure 6 we show the results of both experiments.



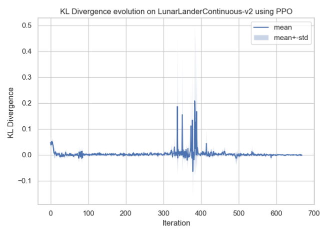
(a) Episode rewards



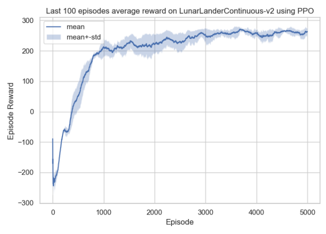
(b) Actor loss



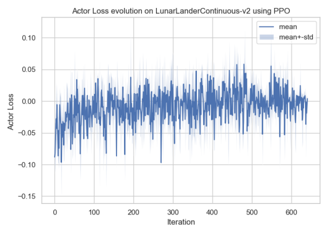
(c) Critic loss



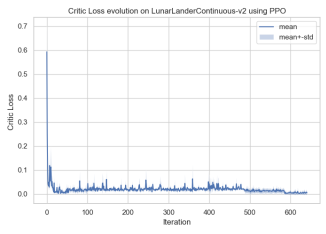
(d) KL-Divergence



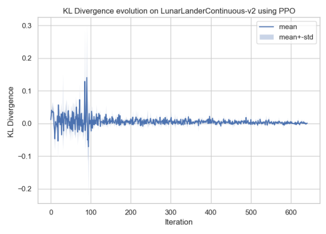
(e) Episode rewards



(f) Actor loss



(g) Critic loss



(h) KL-Divergence

Figure 6: LunarLanderContinuous-v2: Experiment 1 (first row) and 2 (second row) results

The first experiment cannot solve the environment, reaching different average rewards in each repetition. The shaded zone representing the standard deviation is very wide in the last episodes, going down to negative 300 points because the performance crumbles in one of the executions. Looking at the actor loss, we cannot see anything, only that the actor is learning something as the loss is not 0. The critic loss is decreasing, which is the expected trend. However, the range of values is very high, and the clipping to avoid exploding gradients does not allow to learn better. Looking at the KL-Divergence, we see that it takes high values around the 400 iterations, which we believe is the point where the performance collapses. As we avoid updating the actor when the KL-Divergence is above the maximum, the actor is not updated to correct the mistakes, and when it can update again, it is too late to recover.

By re-scaling the rewards and the gradient clipping, we solve the environment in the second experiment with very few episodes, and the performance does not collapse at any repetition. The critic loss has now a very small range, and it is also decreasing without those wide oscillations. Also, the KL-Divergence is always small enough for the actor to update the policy, so PPO can control the updates correctly.

In Figure 7 we show the reward of 100 extra episodes run with the best model weights (i.e. the ones achieving the best average reward of the last 100 episodes). When testing, we do not randomly pick an action using the learnt distribution, but instead select the mean of the distribution, which should have the highest probability. As we can see, all the episodes reach more than 200 points, with an average of 281.67.

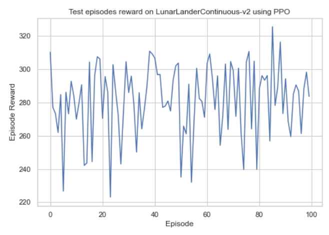


Figure 7: LunardLanderContinuous-v2: Experiment 2 test results

4.2 BipedalWalker-v3

We also performed two experiments, but strictly speaking, we cannot solve the environment in any of them. We solve the environment if we reach an average of 300 over 100 consecutive episodes. In the first, we reach an average of 260 over the three executions and an average of 295∼298 in the second. One of the seconds' repetitions surpasses

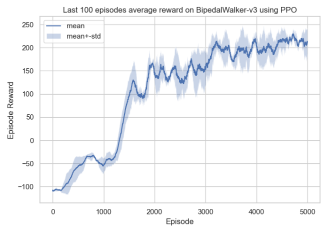
300, but not all three. We believe trying more parameters will solve it, but we left it as future work as we wanted to try more environments with different features.

The parameters of both experiments are listed in Table 2. As we can see, we are using the same parameters as the previous environment. We wanted to see again the importance of the reward scale. It was not that critical, as the rewards we reach during the episodes are usually smaller, ranging from -100 to +100 during the first stages of the learning, and then going up to +300. However, we have observed differences in the way the robot walks. In the second experiment, we get a faster walking that achieves more points.

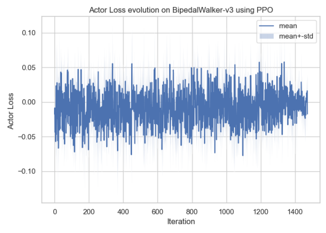
Parameter	Experiment 1	Experiment 2
Reward scale	1	0.01
Buffer size	512	512
Batch size	512	512
Gradient clipping	0.5	50.0
Max KL-Divergence	0.15	0.15

Table 2: BipedalWalker-v3: Experiments hyper-parameters

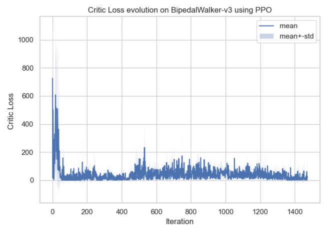
In Figure 8 we show the results of each experiment. As we can see, in the second experiment we run the algorithm for more episodes. We first ran it for 5000 episodes, but we saw the average reward was still increasing, so we increased the number of episodes. In the second experiment, the agent can reach 200 points in very few episodes, needing two times more episodes in the first. Besides, we can see how the critic loss has a small range without scaling than in the previous environment, making it less probable to crumble. If we look at the KL-Divergence, we can see how in most cases the value is not surpassing the maximum, only at the first iterations, which is completely normal.



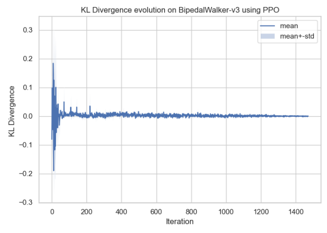
(a) Episode rewards



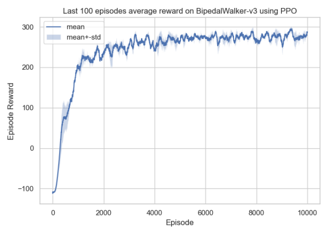
(b) Actor loss



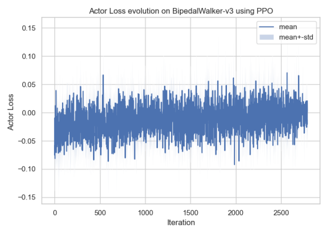
(c) Critic loss



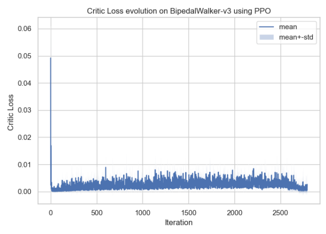
(d) KL-Divergence



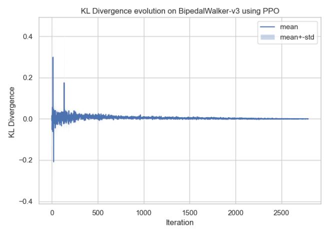
(e) Episode rewards



(f) Actor loss



(g) Critic loss

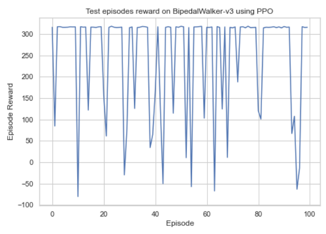


(h) KL-Divergence

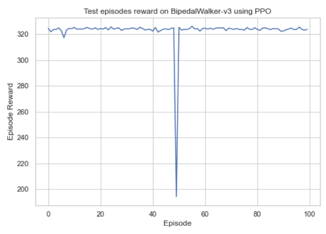
Figure 8: BipedalWalker-v3: Experiment 1 (first row) and 2 (second row) results

In Appendix A, we show some frames from a random episode executed with the best weights of each experiment to show how the robot has learnt to walk. In the first experiment, the agent has learnt to crawl one leg to reach the end as safe as possible, which is not the fastest way nor the safest, as the agent still falls. In the second, the agent has learnt to use both legs to walk faster and safer, as it falls less.

Finally, in Figure 9, we show the results of executing the agent of each experiment for 100 episodes. Again, when selecting the action, we choose the mean of the distribution instead of randomly sampling from it. The results of the first experiment reach more than 300 points in 74 of the 100 episodes. Above the remaining 26, some get negative reward, meaning that the robot does not progress a lot in the course. The average reward is 247.17, meaning that the agent cannot solve the environment once we remove the exploration. In the second experiment, we can see how the agent surpasses the 300 points in all the episodes except in one (getting more than 200 points), reaching an average reward of 322.81, meaning that the agent solves the environment without exploration.



(a) Experiment 1



(b) Experiment 2

Figure 9: BipedalWalker-v3: Experiment 1 and 2 test results

4.3 basic

ViZDoom does not define solving, but for this one, reaching an average of more than 80 points should be enough. Therefore, we were able to "solve" it on the first try.

However, we had to apply some pre-processing tricks to feed the frames/states to the neural network. In particular, we start by not rendering the HUD when training, converting the RGB frames to grey-scale, resizing them to 100x100 pixels, and normalizing the intensities. Then, to generate the state that enters the network, we stack 4 "consecutive" frames to add the temporal information of the displacement. Finally, when executing an action, we repeat it 4 times to avoid processing almost equal frames (i.e. we skip frames). We also apply it in *health_ gathering*, so we do not explain it there.

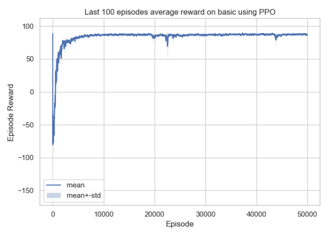
We present the hyper-parameters in Table 3, which are almost the same as with the previous environments, only changing the gradient clipping. As the state is composed of frames, and we were not sure about the right value, we set it quite low to avoid exploding gradients.

Parameter	Value
Reward scale	0.01
Buffer size	512
Batch size	512
Gradient clipping	0.5
Max KL-Divergence	0.15

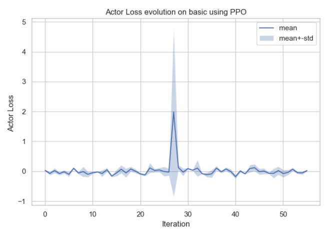
Table 3: *basic* hyper-parameters

The network architecture, shared with *health_ gathering*, it is composed of 3 convolutional layers with 32, 64, and 128 filters, ReLU activation, each followed by an average pooling. After the flatten layer, we have a dense with 256 units and ReLU activation, and the final softmax layer to output the action probabilities.

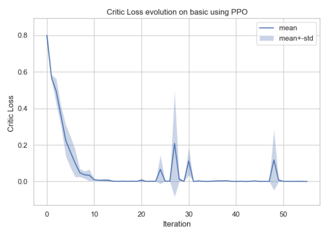
In Figure 10 we show the training results. We decided to run 50000 episodes because each episode runs pretty fast, and we were not sure about how much it would take to converge. As we can see, from 10000 episodes on-wards, the average reward slightly changes, only in those places where the agent still fails to kill the monster, getting a negative reward, and those peaks in the other plots. However, we can overall see good stability.



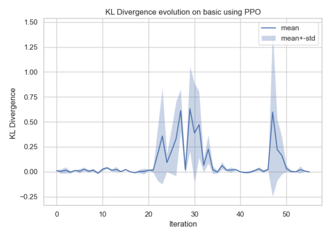
(a) Episode rewards



(b) Actor loss



(c) Critic loss



(d) KL-Divergence

Figure 10: *basic* results

In Figure 11 we show 100 extra episodes executed in test mode. As we can see, we are reaching the maximum points (95) half of the time, and then is quite variable, mostly above 70. We end up with an average reward of

86.07, which we can consider also as “solved”.

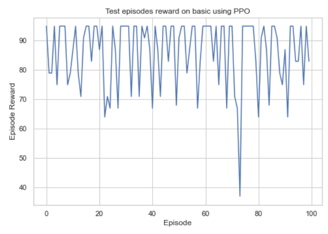


Figure 11: *basic* test results

4.4 health_gathering

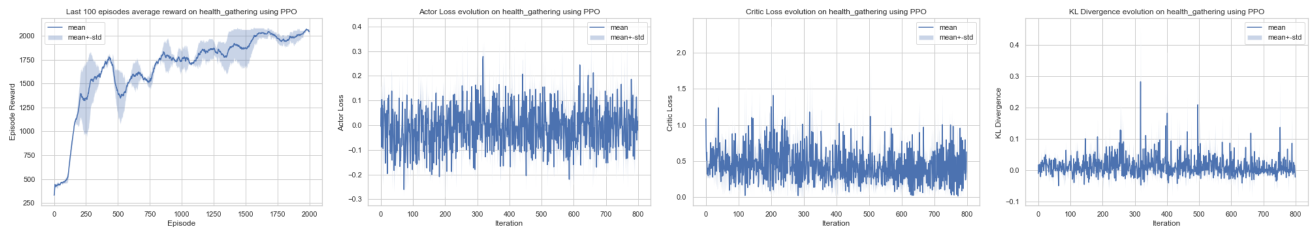
For this environment, we do not have any definition of solving, but we can suppose reaching an average of more than 2000 points in the last 100 episodes is pretty close to reaching the perfect score. And based on this definition, we are able to also “solve” it in our “first” try.

In Table 4 we show the hyper-parameters, which are quite different than those used in *basic*. We reduced the buffer size and batch size because as the agent easily learns to complete some episodes (i.e. reaching the maximum time steps), the training can become very slow otherwise. Also, we have incremented the maximum KL-Divergence allowed because we ran it once with the value of 0.15 for one repetition, and in most iterations we surpassed that value. Increasing it to 0.25 allows us to learn better.

Parameter	Value
Reward scale	0.01
Buffer size	128
Batch size	64
Gradient clipping	0.5
Max KL-Divergence	0.25

Table 4: *health_gathering* hyper-parameters

In Figure 12 we show the training results. As we can see, we have only trained for 2000 episodes, and it reaches the desired average in all repetitions, but with some variance in the first stages of learning. We leave as future work training for mode episodes and try to maximize the score. We can see some oscillations in the actor and critic loss, but the range is really low. Finally, if we look at the KL-Divergence, we see that with the new value, we do not surpass the maximum value in most cases, not even the previous 0.15, confirming that we are learning better.



(a) Episode rewards (b) Actor loss (c) Critic loss (d) KL-Divergence

Figure 12: *health_gathering* results

Finally, in Figure 13 we show 100 extra episodes run in test mode. As we can see, we are reaching the maximum score in all the episodes.

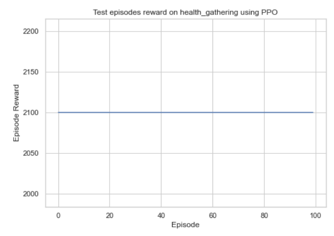


Figure 13: *health_ gathering* test results

5 Conclusions

In this assignment, we have implemented PPO, one of the state-of-the-art deep reinforcement learning algorithms, and tested it in environments with different features. We have not been able to test it in all the environments we would like to, like some Atari games, which results in future work to continue someday. Also, we would like to apply some of the concepts of the second part, like using intrinsic rewards to solve certain environments. However, we have been able to put into practice the different concepts learnt during the first part of the subject and struggle to train the agents to reach the best possible results in the different environments. We have not achieved the best results in some cases, but overall the work has been fun and satisfactory.

References

[1] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[2] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. DOI: 10.48550/ARXIV.1506.02438. URL: https://arxiv.org/abs/1506.02438.

[3] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[4] Yuhui Wang, Hao He, and Xiaoyang Tan. "Truly Proximal Policy Optimization". In: *CoRR* abs/1903.07940 (2019). arXiv: 1903.07940. URL: http://arxiv.org/abs/1903.07940.

A Appendix: BipedalWalker-v3 frames

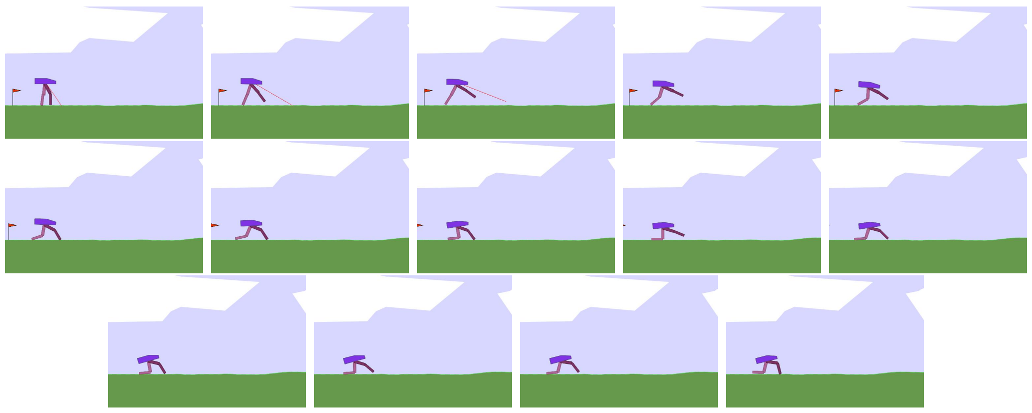


Figure 14: BipedalWalker-v3: Experiment 1 episode frames

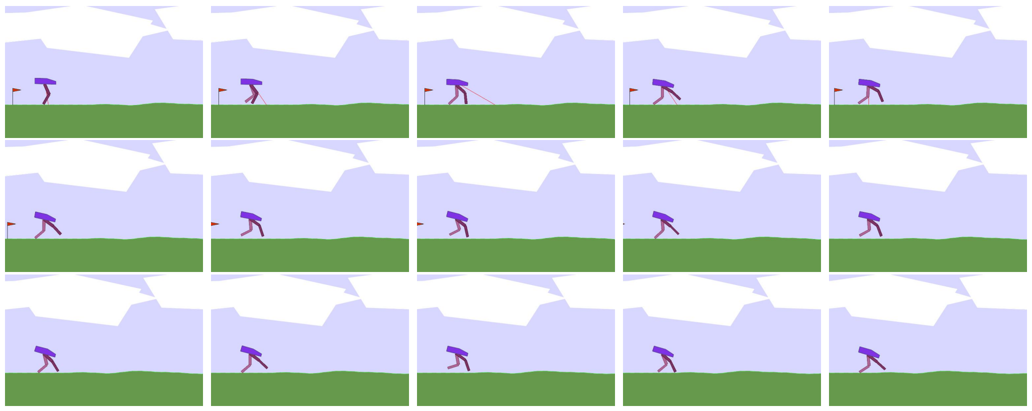


Figure 15: BipedalWalker-v3: Experiment 2 episode frames