Advanced Topics in Computational Intelligence

Proximal Policy Optimization

Author:

González Béjar, Javier

ATCI: Practical Project 2023/24

Master's degree in Artificial Intelligence

Barcelona, May 16, 2024

Contents

1 Introduction

This work consists on the implementation of the PPO algorithm, described in the paper **Proximal Policy Optimization Algorithms**[5], by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov. The algorithm is a policy gradient method used in reinforcement learning which innovates by using a novel objective function that enables multiple epochs of minibatch updates. Unlike traditional policy gradient methods, which perform a single update per data sample, PPO enables a more efficient use of data through its ability to reuse samples for multiple updates, which can lead to faster learning rates and improved stability.

To evaluate the effectiveness of our implementation, we conducted several experiments across various OpenAI Gym environments. In Section 2, we provide a general description of the algorithm. Section 3 is dedicated to detailing our implementation approach. Finally, in Section 4, we first introduce the environments on which our experiments were conducted, followed by a presentation of the experimental configurations used and the results obtained.

2 Algorithm Description

Proximal Policy Optimization (PPO) is an on-policy and actor-critic algorithm. The core idea behind it is to combine the best of both worlds from policy optimization: it aims to achieve the data efficiency and reliable performance of Trust Region Policy Optimization (TRPO) while being significantly simpler to implement and general enough to handle different types of policy networks.

As an on-policy method, PPO learns from the data generated by the current policy being optimised, requiring new data after each training update to ensure that the policy remains relevant to the task. Unlike methods that derive policies from state-value $V(s)$ or state-action value functions $Q(s, a)$, actor-critic approaches learn the policy directly. In this framework, the "actor" network decides actions based on the current policy, and the "critic" network evaluates these actions by estimating the value function, thereby informing the actor's updates and improving its decisions.

PPO employs Generalized Advantage Estimation (GAE) [1] to compute advantages, which measure how much better an action is compared to the policy's average. This advantage estimation helps in reducing the variance of the policy gradient estimates, improving the stability and speed of learning.

$$A_t^{GAE(\gamma,\lambda)} = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} \tag{1}$$

The main innovation in PPO is its objective function, which incorporates a clipping mechanism to avoid large policy updates that could degrade performance. This clipping modifies the policy optimisation objective by limiting, or "clipping", the ratio of the new policy probability to the old policy probability to be within a predetermined interval, $[1 - \epsilon, 1 + \epsilon]$, where ϵ is a small positive value, typically 0.1 or 0.2.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2}$$

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \tag{3}$$

Equation [2] represents the probability ratio of the chosen action a_t given state s_t under the current policy π_θ relative to the probability under the old policy π_{old}. Equation [3] represents the final loss function for the actor, incorporating the advantage and the probability ratio. As we explained, the clipping mechanism in the function helps maintaining a good trade-off between exploring new policies and exploiting the current one. It ensures that the updates are significant enough to make progress, but not too drastic to destabilise the learning process. This "clipping" effect can be seen in the image below, extracted from [5]:

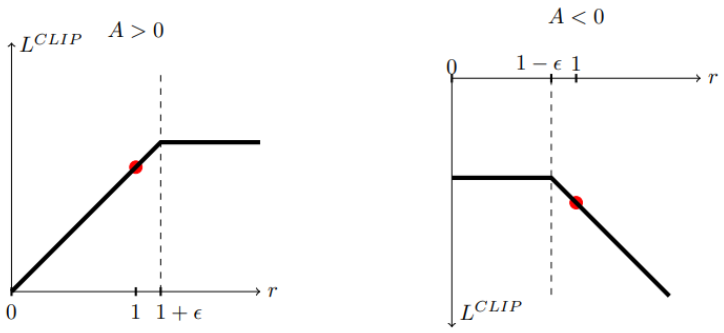


Figure 1: Clipping effect

When the advantage A is positive (left plot), the clipping restricts the probability ratio r to a maximum value of $1+\epsilon$. This prevents the policy from becoming overly aggressive in updating actions that already have high advantages. However, when the advantage A is negative (right plot), the clipping restricts the probability ratio r to a minimum value of $1-\epsilon$. This prevents the policy from significantly reducing the probability of actions that have lower advantages, avoiding overly conservative updates.

Below we find the pseudo-code [1] of PPO, as illustrated in [5].

Algorithm 1 PPO, Actor-Critic Style

for iteration $= 1, 2, \ldots$ **do**
  for actor $= 1, 2, \ldots, N$ **do**
    Run policy π_{old} in environment for T timesteps
    Compute advantage estimates $A_1, \ldots, A_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

The algorithm represents the core steps of the Proximal Policy Optimization in an Actor-Critic style. It iterates over two nested loops, with each iteration involving parallel data collection by multiple actors. Each actor runs the current policy in the environment for T timesteps, collecting data and computing advantage estimates $A_1, \ldots, A_T$. These estimates guide the optimisation of the surrogate objective function with respect to policy parameters θ over K epochs, using minibatches of size M. Finally, the policy parameters θ_{old} are updated to the newly optimized θ, and the process repeats for the next iteration.

3 Implementation

The algorithm is implemented in Python 3.9 using TensorFlow and Keras 2.13.

Initially, some master script files are prepared where we set up OpenAI environments for each experiment we will be performing. Within these files, we define the hyperparameters to be used for each experiment and set up directories for saving data, such as rewards, learning steps, and the number of time steps per episode. The primary execution loop iterates through each episode, which is basically a sequence of interactions between an agent and the environment, where a heuristic (the PPO algorithm) is applied to choose actions at each step, and the rewards from each step are collected.

We decided to fix a maximum number of 5000 episodes for experiences, stopping before if the agent converges to the optimal reward achievable in the environment. Below we find the training pseudo-code [2]:

Algorithm 2 Proximal Policy Optimization Training Procedure

```
Initialise the environment, model configuration and hyperparameters
Prepare directories for model checkpoints and saving results
Initialise agent
for i = 1 to n_games do
    Reset the environment and get initial observation
    while not done do
        Select action using the policy from current observation
        Execute action in the environment and observe the result
        Store transition in the buffer
        if time to update then
            Perform learning step
        end if
        Update observation
    end while
    if average score is improved then
        Update the best score
    end if
end for
Close the environment
```

The PPO implementation centers around a main class named **PPOAgent**, which serves as the base class for the subclasses **PPOAgentDiscrete** and **PPOAgentContinuous**, serving for discrete and continuous action spaces, respectively. The most relevant attributes initialised in the base class include:

- **Number of actions:** Corresponds to the number of actions that are allowed in the environment's action space. This directly influences the output layer of the Action Networks, as they define the probabilities of choosing from these actions in the episodes.
- **Input dimensions:** Defines the observation space of the environment and also sets the input shape for the dense layers in both the Action and Critic networks.
- **Gamma:** The discount factor used to balance immediate and future rewards.
- **GAE Lambda:** The lambda parameter for Generalized Advantage Estimation, important for calculating the advantage metric during training.
- **Number of epochs:** Specifies how many training iterations the network undergoes during each learning phase.

Other methods in **PPOAgent** are used to ease model saving and loading, and manage the storage of episode transitions within a buffer. We also implemented a **learn()** function [3], which is particularly important for the agent learning. It manages the training process by using stored transitions to improve agent performance.

Algorithm 3 Learn Function for PPO Agent

for each epoch from 1 to n_epochs **do**
  states, actions, log_probs_old, values, rewards, dones, batches ← buffer.generate_batches()
  Initialise advantage
  for $t = 0$ to length of rewards - 2 **do**
    Initialize discount = 1 and $a_t = 0$
    for $k = t$ to length of rewards - 2 **do**
      a_t += discount × (rewards[k] + γ × values[$k+1$] × (1 − dones[k]) − values[k])
      discount *= γ × gae_lambda
    end for
    advantage[t] ← a_t
  end for
  for each batch in batches **do**
    Use gradient tape for automatic differentiation
    Calculate batch_indices, states, actions
    update Actor and Critic networks and get losses
    Compute gradients and apply them to actor and critic
  end for
  Clear the buffer
end for

The function performs the following steps:

1. **Generate batches of transitions:** Compiles groups of experiences collected during interactions with the environment.

2. **Calculate the advantage using GAE:** Determines the relative value of each action taken in the context of subsequent states and rewards.

3. **Update the networks using the batches and the advantage:** Applies gradient updates to the Actor and Critic networks to refine their predictions and value estimations.

4. **Clear the buffer:** Resets the storage buffer to make space for new data in following episodes.

During each epoch of learning, we are processing batches of transitions to update network parameters. For each batch, the **tf.GradientTape** context is used to dynamically calculate gradients for optimising the networks. These gradients are then applied to adjust the network weights with the goal of reducing differences between predicted and actual outcomes, consequently fine-tuning the agent's decision-making ability.

For the discrete agent, when we update the networks, we use the actor's outputs to create a categorical distribution, calculating logarithm probabilities of the actions taken. The probability ratio between new and old log probabilities is used with the PPO clipping technique to compute the actor loss, which includes an entropy component to promote exploration. The critic's loss is computed using mean squared error to evaluate the accuracy of value predictions.

For the continuous agent, on the other hand, the actor network generates means and standard deviations to define a normal distribution for action probabilities. After computing logarithm probabilities, the same clipping strategy is applied.

The last relevant method on the class is **choose_action()**, which also works differently based on the agent's action space. For discrete action spaces, it converts the observation into a tensor, derives action probabilities from the actor network, and selects an action from a categorical distribution, also calculating the action's log probability and state value from the critic network. For continuous action spaces, the method processes observations through the actor to obtain parameters for a normal distribution, samples an action, and computes its log probability and state value, making sure actions cover a continuous range to support dynamic policy updates during training.

This 'continuous' method of action selection, reward evaluation, and network updates is the main part of the PPO agent's learning mechanism, and is what allows it to progressively improve and adapt to the complexities of the specified environment.

Finally, we present two more code files, **PPONetworkDiscrete** and **PPONetworksContinuous**, which contain Actor and Critic Networks for Discrete and Continuous action spaces.

In the first case, the Actor Network outputs a probability distribution over possible actions through a 'softmax' layer following two ReLU-activated dense layers, used for selecting one among several

distinct actions. The Critic Network, structured similarly but ending in a single output neuron, estimates the state value to assess potential returns.

In the second case, the actor produces parameters for a normal distribution, so it eases action selection across a continuous range, through layers that output means and standard deviations. The critic also uses this setup in its structure but it only outputs a scalar state value estimate.

4 Experimental Section

In this section, we conduct experiments in various OpenAI Gym environments [4], utilising both discrete and continuous action spaces to evaluate the algorithm's effectiveness and efficiency. In total, three different environments are tested, including two discrete environments: CartPole-v1 and Acrobot-v1, and one continuous environment: Pendulum-v1. We analyse and present their results, including average rewards obtained, the number of episodes, and time steps.

4.1 CartPole-v1

The CartPole environment consists of a cart that moves along a one-dimensional track. A pole is attached to the cart by an un-actuated joint. The objective is to prevent the pole from falling over by applying forces to the cart to keep it balanced. The environment is considered solved when the pole remains upright for a specified duration [1].

The action space is discrete, with two possible actions:

- **Push Left (0):** Apply a force to the cart towards the left.
- **Push Right (1):** Apply a force to the cart towards the right.

The state space of the CartPole environment, which is continuous, on the other hand, is represented by a four-dimensional vector, which indicates:

- **Cart Position**: The position of the cart on the track.
- **Cart Velocity**: The velocity of the cart.
- **Pole Angle**: The angle of the pole with the vertical axis.
- **Pole Angular Velocity**: The rate of change of the pole's angle.

The goal is to keep the pole upright for as long as possible, and the agent is given a +1 reward for each step taken. The environment is considered solved when the agent accumulates a total reward of 500, which corresponds to keeping the pole upright for 500 steps.

In this environment, we will be running 6 different experiments to evaluate the performance of our agent with different configurations. Each experiment will run for up to 5000 episodes or until the average reward of the last 20 episodes reaches 500, indicating consistent optimal performance.

In Tables 1 and 2, we find the parameter configurations for each experiment performed. The common hyperparameters, which have yielded good results in state-of-the-art research with PPO, are set as follows: the batch size is 5, the number of epochs is 4, the learning rate is 1e-3, the discount factor γ is 0.99, and the GAE λ is 0.95. The specific configuration grid for the experiments is as follows:

Parameter	Experiment 1	Experiment 2	Experiment 3	Experiment 4
batch_size	5	5	5	5
n_epochs	4	4	4	4
learning_rate	0.001	0.001	0.001	0.001
discount_factor	0.99	0.99	0.99	0.99
gae_lambda	0.95	0.95	0.95	0.95
clip_ratio	0.1	0.1	0.1	0.1
c1 (vf_coef)	0.5	0.5	1	1
c2 (ent_coef)	0.01	0.001	0.01	0.001

Table 1: Parameter configuration for each experiment (Part 1)

Parameter	Experiment 5	Experiment 6	Experiment 7	Experiment 8
batch_size	5	5	5	5
n_epochs	4	4	4	4
learning_rate	0.0001	0.0001	0.0001	0.0001
discount_factor	0.99	0.99	0.99	0.99
gae_lambda	0.95	0.95	0.95	0.95
clip_ratio	0.2	0.2	0.2	0.2
c1 (vf_coef)	0.5	0.5	1	1
c2 (ent_coef)	0.01	0.001	0.01	0.001

Table 2: Parameter configuration for each experiment (Part 2)

In Figure 2 we show the results of these experiments.

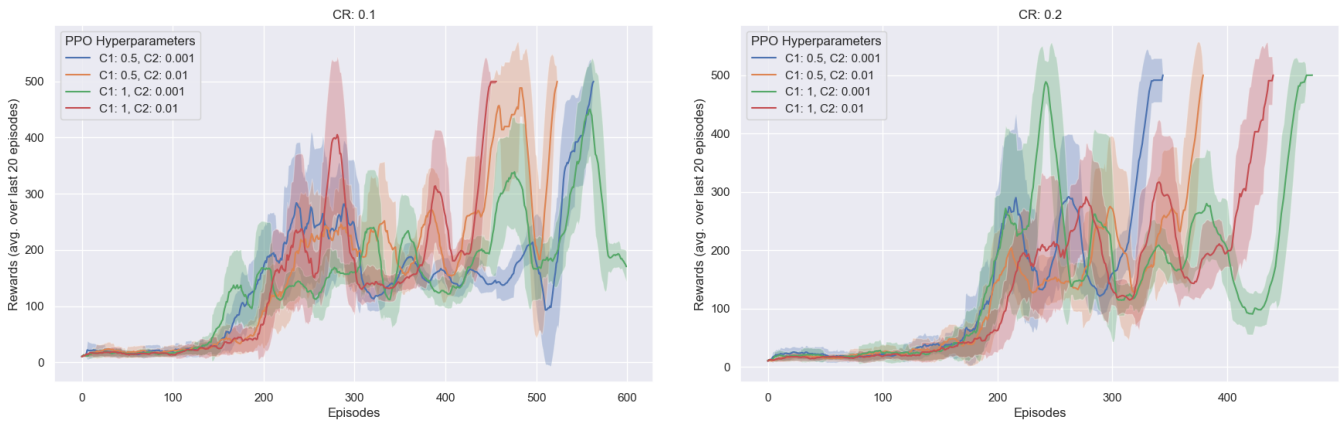


Figure 2: CartPoleV1: Mean Reward [20 window] per episode - Hyperparameters GridSearch. CR stands for ϵ (Clip Ratio), C1 for Value Coefficient, C2 for Entropy Coefficient

The experimental results above, from the CartPole-v1 environment, reveal significant volatility in rewards across episodes for all tested hyperparameter configurations, which indicates substantial variation in performance. Configurations with a higher C2 value tend to show more volatility and converge sooner (they arrive to the desired peak average reward before than other configurations), suggesting that higher entropy coefficients may promote more explorative behaviours, but also increase performance inconsistency.

When we compare different clip ratios (CR: 0.1 vs. CR: 0.2), it can be observed that a higher clip ratio can potentially accelerate learning but also increase the probability of having extreme performance swings, which reflects the trade-off between speed of learning and stability.

It can also be inferred that configurations with a higher value coefficient (C1: 1.0) start with lower rewards, but they also show significant improvement over time, implying that they may require a longer learning period to learn, but could achieve more stable results in the long term.

Overall, the experiments highlight the sensitivity of the CartPole-v1 environment to specific PPO hyperparameters like the value and entropy coefficients, which means that their tuning is specially important if we want to optimise both their performance and consistency.

Below, we plot the model with the following configuration: **CR:01, C1:1, C2:0.001**. It took 2464 episodes to converge, showcasing the worst performance of the evaluated experiments.

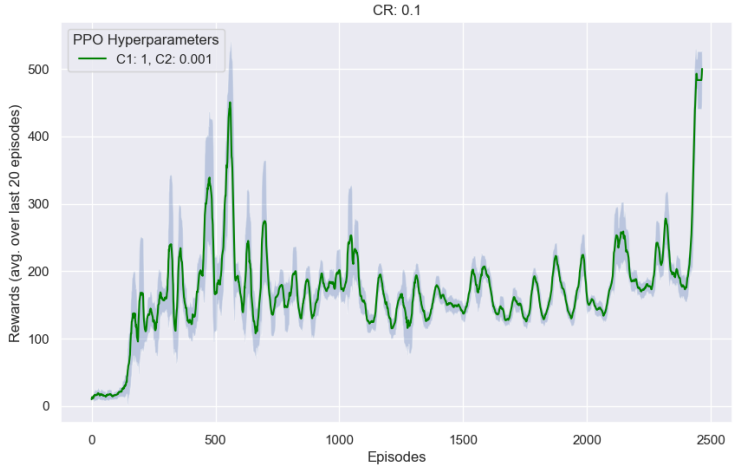


Figure 3: Worst performing Model Training: ϵ : 0.1, C1: 1, C2: 0.001

Finally, we present detailed results in the tables below (Tables 3 and 4). These tables concretely detail the number of episodes trained, mean rewards, timesteps, and occurrences of truncations or terminations.

ϵ	C_1	C_2	Episodes	R_{20}	R_{avg}	T_{avg}	Terminations	Truncations
0.1	0.5	0.001	564	500.0	151.48	151.48	535	29
0.1	0.5	0.01	524	500.0	171.55	171.55	469	55
0.1	1.0	0.001	2465	500.0	172.69	172.69	2403	62
0.1	1.0	0.01	457	500.0	144.03	144.03	419	38

Table 3: Performance metrics across different configurations with ϵ set to 0.1.

ϵ	C_1	C_2	Episodes	R_{20}	R_{avg}	T_{avg}	Terminations	Truncations
0.2	0.5	0.001	345	500.0	136.00	136.00	313	32
0.2	0.5	0.01	380	500.0	124.48	124.48	354	26
0.2	1.0	0.001	475	500.0	157.61	157.61	421	54
0.2	1.0	0.01	441	500.0	142.35	142.35	409	32

Table 4: Performance metrics across different configurations with ϵ set to 0.2.

We can infer from the table results along the graphs presented before that configurations with a higher ϵ value show faster but less stable performance, while lower ϵ values are associated with more consistent and higher average rewards over longer episodes, suggesting a trade-off between stability and the speed of convergence in learning performance.

4.2 Acrobot-v1

The Acrobot environment consists of a double pendulum system where the objective is to swing the lower pendulum upwards until both pendulums are vertically aligned with the base. This environment is considered solved when the top pendulum reaches a position above a horizontal line at the base [3].

The action space is also discrete and deterministic, now with three possible actions that represent the torque applied on the actuated joint between the two links:

- **Negative Torque (0)**: Apply a negative torque to the joint.
- **No Torque (1)**: Apply no torque.
- **Positive Torque (2):** Apply a positive torque to the joint.

The state space of the environment, continuous, is represented by a six-dimensional vector indicating:

- **cos**(θ_1): The cosine of the angle of the first pendulum with the vertical.
- **sin**(θ_1): The sine of the angle of the first pendulum with the vertical.

- **cos(θ_2)**: The cosine of the angle of the second pendulum with the vertical.
- **sin(θ_2)**: The sine of the angle of the second pendulum with the vertical.
- The angular velocity of the first pendulum.
- The angular velocity of the second pendulum.

In this case, the main objective is to swing the lower pendulum upwards until both pendulums are vertically aligned with the base. The agent is given a reward of -1 for each step taken until the target position is reached. Achieving this target height results in termination with a reward of 0.

To train our algorithm, we plan to run 1,000 different episodes. Initially, we considered running 5,000 episodes; however, this specific environment was causing memory issues, leading to system crashes shortly after reaching 1,000 episodes.

Below, Table 5 presents the configuration of hyperparameters that yielded the best results in previous discrete experiment. We have also experimented with other configurations that produced similar or inferior performance, although they are not included in the report. The learning rate and number of epochs have been increased to 0.001 and 16, respectively, to align with research that indicates that these hyperparameters typically yield good results in this environment.

Parameter	Value
batch_size	5
n_epochs	16
learning_rate	0.001
discount_factor	0.99
gae_lambda	0.95
clip_ratio	0.2
c1 (vf_coef)	0.5
c2 (ent_coef)	0.001

Table 5: Parameters for Acrobot Experiment

Now we present below in Figure r4 the results of training the model, plotting number of episodes against average rewards (with a 20 reward window to smooth the plot).

Figure 4: Acrobot-v1: Mean Reward [20 window] per episode - Agent training. CR stands for ϵ (Clip Ratio), C1 for Value Coefficient, C2 for Entropy Coefficient

The data from Figure 4 and Table 6 reveals a pattern of initial reward instability that stabilises over time. The performance graph proves that the agent, while initially experiencing sharp fluctuations, progressively improves, suggesting effective learning despite the challenges of this environment. However, we can't consider it's strictly solved, as it should be stabilising in 0 reward.

Over 991 episodes, the rewards gradually increased, reducing the magnitude of losses from -106.88 on average to -82.05 for the last 20 episodes. This, along with the growing slope suggest that there's still room for improvement and the scenario would have probably fully converged if the agent continued learning.

The agent, as seen in the table, is failing to solve the environment most times, as indicated by 971 terminations. These results underscore the need for further training (1000 episodes is probably a short learning) or parameter optimisation or exploration of alternative strategies, given the complexity of this Acrobot-v1 environment.

ϵ	C_1	C_2	Episodes	R_{20}	R_{avg}	T_{avg}	Terminations	Truncations
0.2	0.5	0.001	991	-82.05	-106.88	107.86	971	20

Table 6: Performance metrics for the experiment, with ϵ set to 0.2, C_1 to 0.5, and C_2 to 0.001.

4.3 Pendulum-v1

The Pendulum environment consists in a simple pendulum, including a pole attached to a fixed pivot point without any motion. The objective is to apply torque to the pivot to swing the pendulum upward so that it stays upright. Unlike some environments where the goal is to maintain stability passively, in Pendulum-v1, the challenge is to reach and maintain a vertically upright position [2].

The environment is considered solved when the pendulum remains upright with minimal oscillation

for a certain duration, being the maximum reward 0.

The action space in Pendulum-v1 is continuous, representing the:

- **Torque applied:** The torque can vary continuously, allowing control over the pendulum's movement.

The state space, which is also continuous, is represented by a three-dimensional vector, indicating:

- **Cosine of the angle**: Cosine of the angle between the pole and the vertical axis to help determine the pole's current angle.
- **Sine of the angle**: Sine of the angle, providing a complete representation of the pole's angle.
- **Angular velocity**: The rate at which the angle is changing, which is important for applying the correct amount of torque.

As we just explained, the goal is to swing the pendulum up and keep it upright by applying appropriate torques based on the pendulum's current state. The agent receives a reward each step based on how close the pendulum is to the upright position and the smoothness of its movement. This environment is particularly challenging, more than the Acrobot one, due to its continuous action space.

Despite conducting several experiments with different parameter configurations, this environment has not yet yielded the best possible results due to difficulties in achieving convergence. We have used some optimal configurations identified in updated research, as well as the best-performing ones discovered through grid searches. From our latest experiments, we have reduced the learning rate to 0.0003, increased the number of epochs to 16, expanded the batch size to 64, and set the entropy coefficient to zero [7]. This is because we found out that minimising entropy regularisation helps stabilise the training process by reducing the variability in policy updates.

Table 7: Parameters for Experiment 1

Parameter	Value
batch_size	64
n_epochs	16
learning_rate	0.0003
discount_factor	0.99
gae_lambda	0.95
clip_ratio	0.2
c1 (vf_coef)	0.5
c2 (ent_coef)	0

Below, Figures 5 and 6 illustrate the results of our training with the specified hyperparameters. Initially, we observed that the training not only failed to converge but also showed significant instability. Consequently, we implemented gradient clipping of 50 and reward scaling of 0.01, adjustments supported by research indicating potential performance improvements.

These modifications significantly influenced learning, as proved by both experiments, which consistently showed deeply negative rewards over 5800 episodes, underscoring the difficulty of the task.

The first experiment shows a modest improvement in average rewards towards the end, while the second experiment, incorporating a reward rescale factor of 0.01 and gradient clipping of 50, achieved a noticeably better overall average reward. This suggests that these adjustments effectively mitigate the impact of extreme negative rewards and stabilise the training process. Despite these improvements, the high number of truncations in both scenarios indicates that episodes consistently reach their maximum step count without successfully meeting termination criteria, highlighting ongoing challenges within this environment.

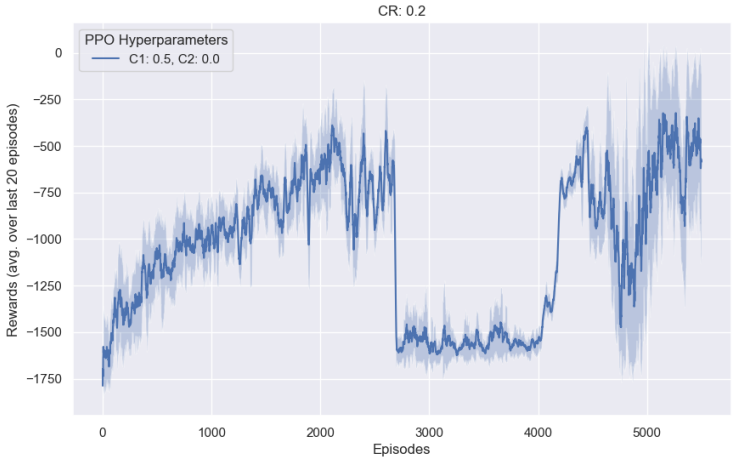
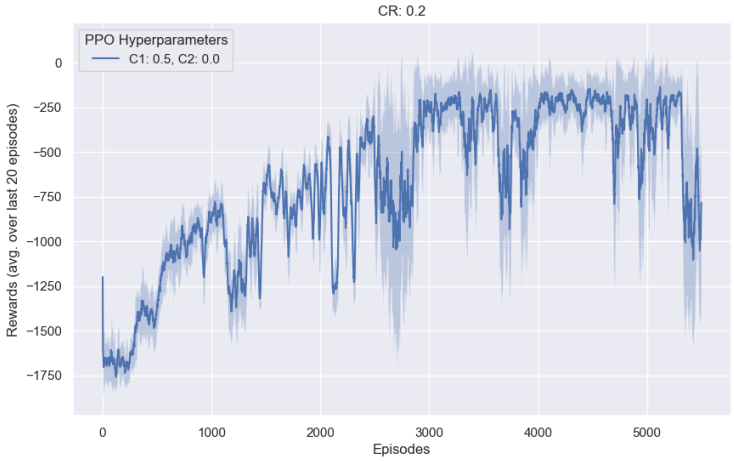


Figure 5: Pendulum-v1: Mean Reward [20 window] per episode - Agent training. CR stands for ϵ (Clip Ratio), C1 for Value Coefficient, C2 for Entropy Coefficient



(a) Standard error results in artificial datasets

Figure 6: Pendulum-v1: Mean Reward [20 window] per episode - Agent training. CR stands for ϵ (Clip Ratio), C1 for Value Coefficient, C2 for Entropy Coefficient. Using reward rescaling of 0.01 and Gradient Clipping of 50

Table 8: Performance metrics for Config 1 with ϵ set to 0.2, C_1 to 0.5, and C_2 to 0.0.

ϵ	C_1	C_2	Episodes	R_{20}	R_{avg}	T_{avg}	Terminations	Truncations
0.2	0.5	0.0	5800	-297.1	-1021.26	200.0	0	5795

Table 9: Performance metrics for Config 1 with ϵ set to 0.2, C_1 to 0.5, and C_2 to 0.0.

ϵ	C_1	C_2	Episodes	R_{20}	R_{avg}	T_{avg}	Terminations	Truncations
0.2	0.5	0.0	5800	-359.77	-676.36	200.0	0	5722

5 Conclusions

In this study, we applied the Proximal Policy Optimization (PPO) algorithm to different settings, focusing on both discrete and continuous environments such as CartPole-v1, Pendulum-v1, and Acrobot-v1.

We observed that the algorithm performed better in discrete environments and struggled to fully converge in continuous ones. Particularly for the Pendulum environment, introducing reward scaling and gradient clipping significantly improved stability and performance. Despite these improvements, the challenges persisted across the tasks, with many episodes failing to meet termination criteria. This difficulty highlights the complexities involved in fine-tuning PPO to achieve optimal results in dynamic environments, emphasising the need for careful selection of hyperparameters to maximise the effectiveness of the algorithm in handling complex control tasks.

References

[1] Farama Foundation. *CartPole - Classic Control*. Gymnasium Classic Control Environments. 2024. URL: https://gymnasium.farama.org/environments/classic_control/cart_pole/.

[2] Farama Foundation. *Pendulum - Classic Control*. 2024. URL: https://gymnasium.farama.org/environments/classic_control/pendulum/.

[3] Farama Foundation. *Acrobot Environment*. 2024. URL: https://gymnasium.farama.org/environments/classic_control/acrobot/.

[4] Farama Foundation. *Gymnasium Environments*. 2024. URL: https://gymnasium.farama.org/environments.

[5] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].