University of Pisa

Symbolic and Evolutionary Artificial Intelligence

Implementation and test of advanced RL techniques in a didactic way

Pardini Marco
Acampora Vittoria

A.Y. 2023-2024

Contents

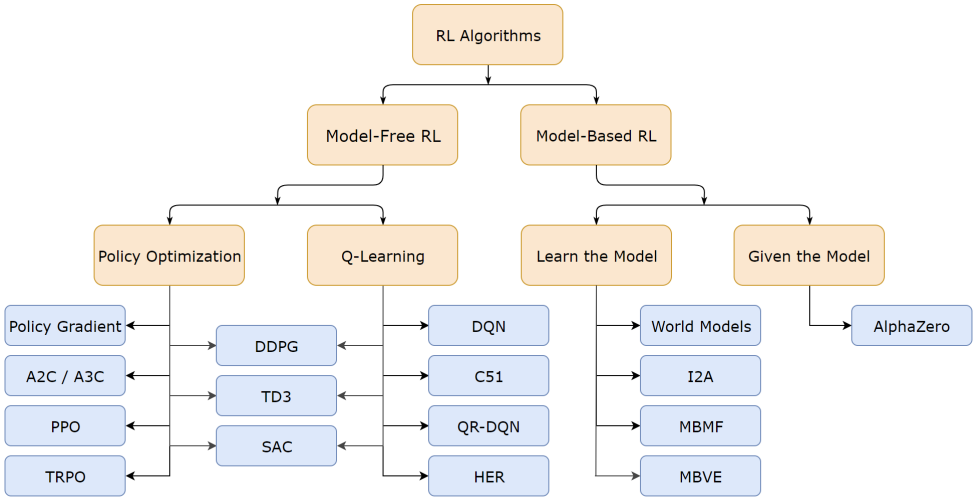
1 Introduction **2**
- 1.1 Model-free algorithms . . . 2
  - 1.1.1 Stochastic Policy optimization . . . 2
- 1.2 Benchmark: CartPole-v1 . . . 3
  - 1.2.1 Solved environment criteria . . . 3

2 Algorithms **4**
- 2.1 Critic Only methods: Q-learning . . . 4
- 2.2 Actor Only methods: REINFORCE . . . 5
  - 2.2.1 Vanilla REINFORCE . . . 5
  - 2.2.2 REINFORCE with baseline . . . 6
  - 2.2.3 Comparison . . . 7
- 2.3 Actor-Critic methods . . . 7
  - 2.3.1 Actor-Critic . . . 7
  - 2.3.2 A2C: Advantage Actor-Critic . . . 9
  - 2.3.3 Comparison . . . 12
- 2.4 PPO: Proximal Policy Optimization . . . 12
  - 2.4.1 The precursor: TRPO . . . 12
  - 2.4.2 PPO: Theoretical formulation . . . 13
  - 2.4.3 PPO implementation . . . 14
  - 2.4.4 Our implementation . . . 15
  - 2.4.5 Results . . . 16

3 Conclusions **19**
- 3.1 Results . . . 19
- 3.2 Next steps . . . 21
- 3.3 Source code . . . 21

Chapter 1

Introduction



1.1 Model-free algorithms

The "Model" in Reinforcement Learning is a representation of the environment (world) and it changes in response to the agent's actions. Model-Free means directly learning from the environment without knowing how it works.
The main advantage of model-free policy learning is that it does not require to identify and parameterize a model of the world (environment) which help you avoid potential errors or restrictions from approximate models.
The downside is that without an explicit environment model, model-free agents must rely on extensive real experience to learn policies (you make the rules). They can be less sample efficient compared to model-based methods that can use planning with a learned world model.

In this paper we're going to review model-free algorithms, and more specifically **Policy Gradient** ones.

1.1.1 Stochastic Policy optimization

The so-called Policy Gradient Algorithms directly manipulate the policy, which in this case is parameterized. It is formalized as $\pi_\theta(a|s)$ with θ representing the parameters to be optimized, and does not require an explicit Value function. Typically, optimization occurs On-Policy, meaning that each parameter's update is performed using experiences obtained by following the latest version of the policy.

1.2 Benchmark: CartPole-v1

For our analysis we choose the benchmark CartPole-v1. The goal of this environment is to balance a pole by applying forces in the left and right directions on the cart. It has a discrete **action space**:

- 0: Push cart to the left
- 1: Push cart to the right

where the action is a ndarray with shape (1,) which can take values 0, 1 indicating the direction of the fixed force the cart is pushed with.

Upon taking an action, either left or right, an agent observes a 4-dimensional state consisting of:

NUM	OBSERVATION	MIN	MAX
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	$\sim$ -0.418 rad(-24°)	$\sim$ 0.418 rad(24°)
3	Pole Angular Velocity	-Inf	Inf

Table 1.1: observation space

where the **observation** is a ndarray with shape (4,).

A **reward** of +1 is granted to the agent at each step while the pole is kept upright. The maximum reward an agent can earn in a single episode is 500.

The **episode ends** under the following conditions:

1. Termination: Pole Angle is greater than $\pm 12°$
2. Termination: Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500

1.2.1 Solved environment criteria

We've decided to consider the *CartPole-v1* environment solved only if we obtain 20 episodes solved in a row. We consider that an episode is solved only if the pole is upright for all the time, which translates in a reward exactly equal to 500.

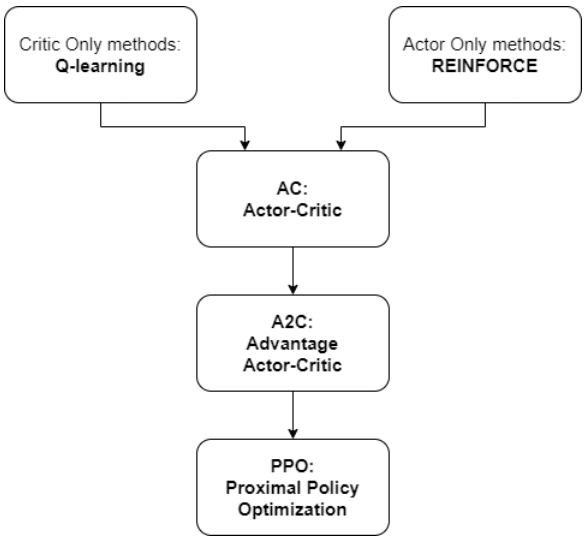
Generally, modern reinforcement learning algorithms such as DQN variants, PPO, and A2C can achieve near-perfect performance on the CartPole benchmark, often balancing the pole for the maximum duration of time allowed.

However this criterion ensures a high level of performance stability from the agent. Achieving 20 consecutive successful episodes demonstrates the ability to balance the pole upright over multiple trials in a consistent way.

Requiring the reward to be exactly 500 ensures that the agent is achieving the best outcome for each episode, rather than simply achieving some level of partial success.

Chapter 2

Algorithms



In this chapter, we will analyze the *Actor-Critic* architecture by starting from its foundational concepts: *Q-learning* and *REINFORCE*. We will explore how these fundamental methods lay the groundwork for the understanding of Actor-Critic methods. From there, we will delve into advanced techniques, culminating in a detailed discussion of *Proximal Policy Optimization (PPO)*, highlighting its improvements and applications within the Actor-Critic framework.

2.1 Critic Only methods: Q-learning

Critic-only methods such as *Q-learning*, *SARSA* and *DQN*, use a state-action value function and no explicit function for the policy. A deterministic policy, denoted by $\pi : X \to U$ is calculated by using an optimization procedure over the value function:

$$\pi(x) = \arg\max_u Q(x, u)$$

In Q-learning and Deep Q Networks (DQNs), incorporating an epsilon-greedy policy is essential for effective exploration. This strategy balances the exploration of potentially valuable but unexplored states (or actions) with the exploitation of known high-value states. By randomly selecting actions with a small probability ϵ (epsilon) and selecting the action with the highest Q-value with probability 1-ϵ, the epsilon-greedy policy ensures a balance between exploration and exploitation.

This approach allows the learning algorithm to gradually transition from exploration to exploitation.

2.2 Actor Only methods: REINFORCE

Policy gradient methods as *REINFORCE* are principally actor-only and do not use any form of a stored value function. The objective is optimizing the cost defined by the average reward

$$J(\pi) = \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left[\sum_{k=0}^{n-1} r_{k+1} \,\middle|\, \pi\right] \tag{2.1}$$

directly over the parameter space of the policy.

A policy gradient method is generally obtained by parameterizing the policy π by the parameter vector $\theta \in \mathbb{R}^p$. Considering that (2.1) are functions of the parameterized policy π_θ, they are in fact functions of θ. Assuming that the parameterization is differentiable with respect to θ, the gradient of the cost function with respect to θ is described by

$$\nabla_\theta J = \frac{\partial J}{\partial \pi_\theta}\frac{\partial \pi_\theta}{\partial \theta}. \tag{2.2}$$

By using standard optimization techniques, a locally optimal solution of the cost J can be found, leading to

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, G_t\right] \tag{2.3}$$

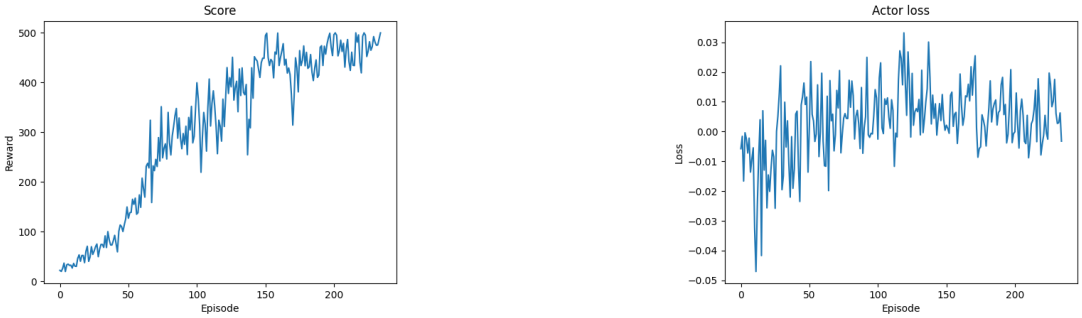
Now that we have outlined the foundational theoretical concepts of Actor-Only methods, we will proceed by examining two implementations of the *REINFORCE* algorithm: the first adheres closely to the above theory, while the second undergoes slight revision.

2.2.1 Vanilla REINFORCE

The classic policy gradient approach, also known as reinforce, uses return G_t to update policy parameters. In this context, G_t represents the sum of the discounted rewards starting from time t.

The advantage of this approach is that it does not require estimation of the value function: it is based only on the observed rewards.

The disadvantages of using G_t is the higher variance in gradient estimates: the sum of rewards from a specific time step t until the end of the episode can be highly variable, especially in environments with high stochasticity or long time horizons. This variance can lead to unstable training and slow convergence.



(a) Score

(b) Policy loss

Figure 2.1: Vanilla REINFORCE

We run Vanilla REINFORCE to solve the *CartPole-v1* environment **with success**.

We set 500 episodes as *max_episodes*, each leading to 10 *trajectories*. This led to a training time total of around 321 seconds, and 234 episodes needed to get to the stopping condition.

Let's see if we manage to obtain a more stable learning process by introducing the *baseline*.

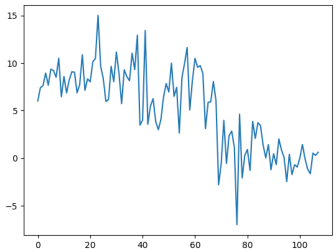
2.2.2 REINFORCE with baseline

The Advantage function attempts to reduce the variance in the gradient estimates. The idea is to use a value function V to normalize the return. The gradient of the cost function J can be rewritten as

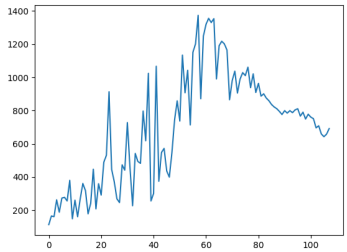
$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, A_\pi(s_t, a_t) \right] \tag{2.4}$$

where $A_\pi(s_t, a_t) = G_t - V_\pi(s_t)$.

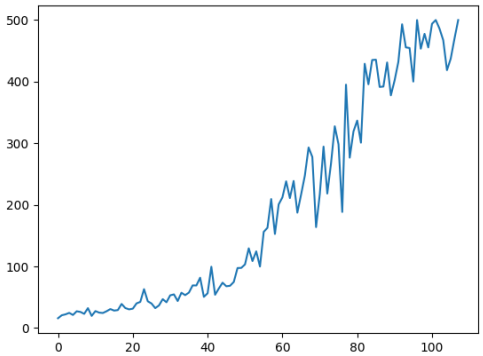
This can be seen as a first step towards the Actor-Critic paradigm, since we now have a parameterized policy π_θ as the Actor, and we have a learned Value function V_π acting as critic. The Value function $V_\pi(s, a)$ will be learned through the recursive relationship dictated by the *Bellman Equations*.



(a) Policy loss



(b) Critic loss



(c) Score

Figure 2.2: PGO with advantage function

We run *REINFORCE with Baseline* to solve the *CartPole-v1* environment **with success**.

As we can see from 2.2c the environment was solved in 108 episodes requiring around 183 seconds. The score convergence seems to be quite smooth and there's nothing particular to underline.

2.2.3 Comparison

If we compare the performances of the 2 methods described above, *Vanilla REINFORCE* and *REINFORCE with Baseline* we can see that introducing the advantage function $A_\pi(s, a)$ proved to be beneficial.

Introducing a baseline in the REINFORCE algorithm accelerated convergence significantly. For instance, the number of episodes needed to achieve the early stopping condition is more than halved with the usage of the advantage function.

This difference will be even more evident with actor-critic methods, that we're going to introduce next.

2.3 Actor-Critic methods

The *Actor-Critic* is a widely used architecture based on the policy gradient theorem.

Actor Critic methods combine the advantages of actor-only methods like *REINFORCE* and critic-only methods like *DQN*. [1]

Those methods can be viewed as a specific instance of the more general *Generative Adversarial Network (GAN)* architecture. In both frameworks, two neural networks are trained concurrently with opposing objectives that drive mutual improvement.

In Actor-Critic methods, the actor network selects actions based on the current state, while the critic network evaluates these actions to provide feedback, thus refining the actor's policy. Similarly, in GANs, the generator network creates data instances to mimic real data, and the discriminator network evaluates these instances to distinguish between real and fake data.

This adversarial training dynamic, where one network's performance directly influences the training of the other, underscores the **problem of convergence**, as both networks must maintain a balance to achieve stable and effective learning outcomes.

In this section we're going to analyze and compare several implementations of the *Actor Critic* paradigm, underlying strengths and issues.

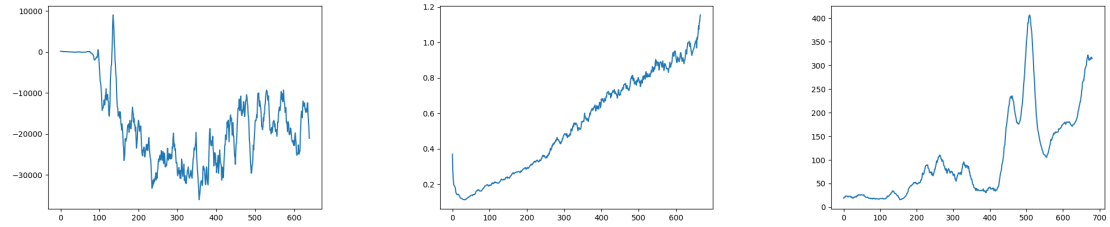
2.3.1 Actor-Critic

The Actor-Critic algorithm leverages the policy gradient theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, Q^\pi(s_t, a_t) \right] \tag{2.5}$$

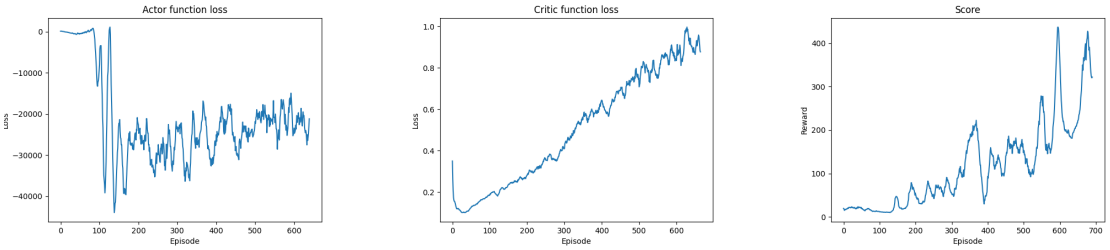
Instead of the unknown true action-value function $Q^\pi(s_t, a_t)$ in equation (2.5), an estimated action-value function $Q^w(s_t, a_t)$ is used, with parameter vector w.

In general, substituting a function approximator $Q^w(s_t, a_t)$ may introduce **bias**. A significant issue with the vanilla Actor-Critic method arises from the absence of Q function normalization through the inclusion of a baseline B, resulting in substantial variance during gradient computation. This variance often obstructs convergence, impeding the algorithm from achieving its optimization goals.



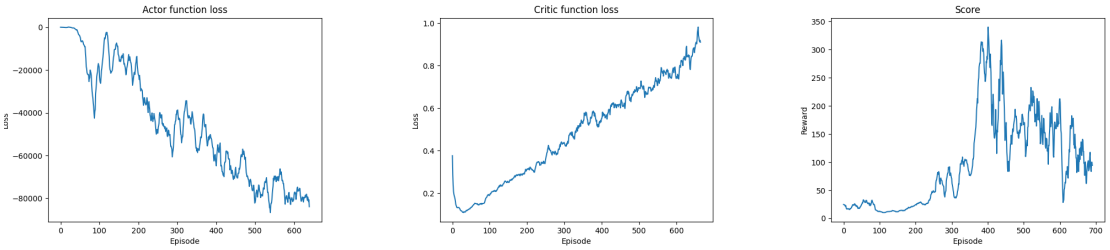
(a) Actor loss (b) Critic loss (c) Score

Figure 2.3: AC first run



(a) Actor loss (b) Critic loss (c) Score

Figure 2.4: AC second run



(a) Actor loss (b) Critic loss (c) Score

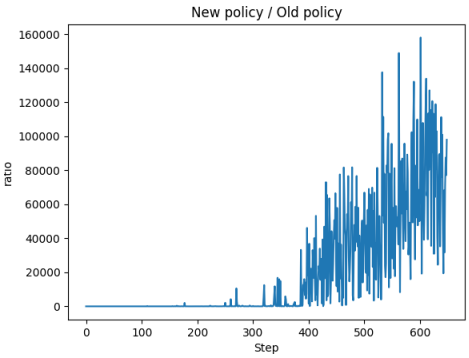
Figure 2.5: AC third run

We run *AC* to solve the *CartPole-v1* environment **without success**. As we can see from 2.3c, 2.4c and 2.5c the algorithm does not manage to achieve stability. Convergence is extremely dependent on the single run, sometimes leading to rewards never reaching the 50 threshold.

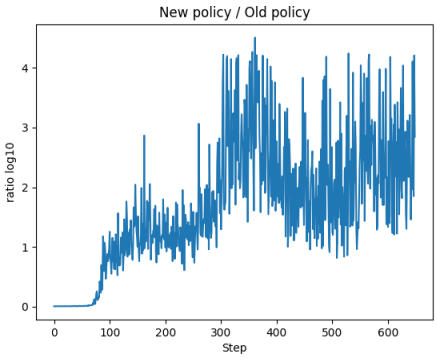
The first plateau in the actor loss shown in 2.3a, 2.4a and 2.5a is due to the need of some episodes for the critic to bootstrap. We empirically observed that if the critic initiates learning simultaneously with the actor, it often results in divergence and the occurrence of exploding gradients, primarily due to the Q function lacking meaningful values. On the other hand, when we limit the training of the critic to only a few initial episodes, we tend to observe a higher likelihood of learning progress, though without achieving convergence.

As we can see in 2.3b, 2.4b and 2.5b, the Critic loss diverges constantly. This likely indicates the fact that the actor policy is constantly changing, never allowing the Q function to learn correctly from extracted samples.

Policy instability discussion



(a) New policy / Old policy



(b) log10(New policy / Old policy)

In order to confirm the policy instability problem in the Actor-Critic algorithm, we decided to plot, at each step, the ratio between the new policy and the old policy.

Algorithm 1 Ratios Computation

```
Input: states_batch, actions_batch, actor_network, old_actor_network

old_distr_params ← old_actor_network(states_batch)
old_m ← Categorical(old_distr_params)
old_log_probs ← old_m.log_prob(actions_batch)

distr_params ← actor_network(states_batch)
m ← Categorical(distr_params)
log_probs ← m.log_prob(actions_batch)

policy_ratios ← torch.exp(log_probs - old_log_probs).mean()
```

As we can see from figure 2.6a starting from episode ≈ 60 ratios starts to diverge, reaching on episode ≈ 600 a ratio approximately equal to 160k times. This means that the log probs produced by the policy at the previous iteration were extremely different from the log probs produced at the successive optimization step. That's a huge indicator that the policy is **extremely unstable**. Later in the paper we're going to analyze the PPO algorithm, that directly tries to overcome this problem.

These behaviours were anticipated, as the standard approach involves implementing A2C directly using the advantage function to mitigate the variance introduced by the Q function.

2.3.2 A2C: Advantage Actor-Critic

Advantage Actor-Critic (A2C) is the evolution of the *Vanilla Actor Critic algorithm*. The key idea is to use the advantage function, which measures how much better an action is compared to the average action, to update the policy more effectively. This results in more stable and efficient learning compared to pure policy gradient methods. A2C enhances performance by **reducing variance** in policy updates while still leveraging the benefits of actor-critic frameworks.

By introducing the *Baseline* we get:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{a \in A} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(Q^\pi(s_t, a_t) - B(s_t) \right) \right] \tag{2.6}$$

Considering only the *Baseline*, if it depends solely on the state S it does not alter the gradient:

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{a \in A} \nabla_\theta \log \pi_\theta(a_t|s_t)\, B(s_t)) \right] = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[B(s_t) \nabla_\theta \sum_{a \in A} \log \pi_\theta(a_t|s_t) \right]$$
$$= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[B(s_t) \nabla_\theta 1 \right] = 0$$

A good choice for *B(s)* is $V_\pi(s)$, so that the *advantage function* becomes:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{2.7}$$

In practice, only $V_\pi(s)$ is used by leveraging *Bellman's Equations*, yielding:

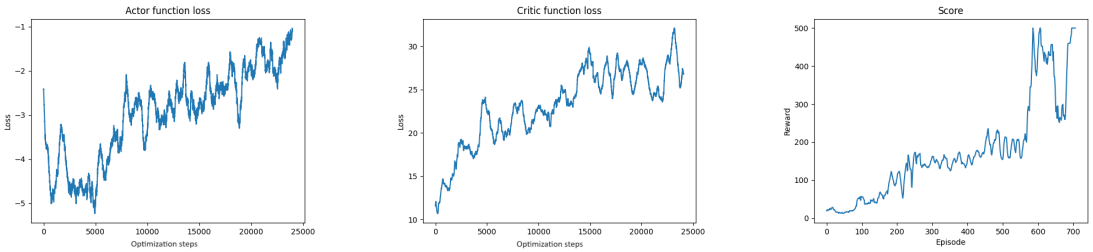
$$A_\pi(s, a) = \mathbb{E}_{s' \in S} \left[r + \gamma V_\pi(s') - V_\pi(s) \right] \tag{2.8}$$

Now, we are going to analyze two variants of *A2C*. The first variant does not utilize target networks for the actor and the critic, relying solely on the current network parameters to update both components.

The second variant incorporates target networks, which are separate copies of the actor and critic networks that are updated less frequently to provide stable targets during training. By comparing these two approaches, we aim to understand the impact of target networks on the stability and performance of *A2C*.

Vanilla A2C

The first variant of A2C we will examine is the *vanilla Advantage Actor-Critic* algorithm, which does not use target networks. In this approach, both the actor, responsible for selecting actions, and the critic, which evaluates the actions, are updated directly based on the latest network parameters. This variant relies on the immediate feedback from the critic to adjust the policy, aiming to optimize performance without the additional stability mechanisms provided by target networks. The simplicity of this method makes it straightforward to implement, but it may suffer from higher variance and less stable learning due to the direct updates.



(a) Actor loss (b) Critic loss (c) Score

Figure 2.7: A2C no target

As we can see from 2.7c the environment was **successfully** solved in 714 episodes requiring around 272 seconds. From figure 2.7a we can see that the trend is to

converge to 0 loss (even if with some noise) while the critic loss function (*MSELoss*) sits around 30. From figure 2.7c we can see that for approximately 600 episodes, the reward remained around 200, and afterwards there's a spike. This may indicate that the policy encountered a plateau which is typical for *policy gradient optimization*, since a small change in the policy's parameter space may lead to huge changes in the policy itself.

Another thing we may notice is that the convergence is quite noisy, and the standard deviation is visibly high, indicating that learning stability is not so satisfying.

A2C with target networks

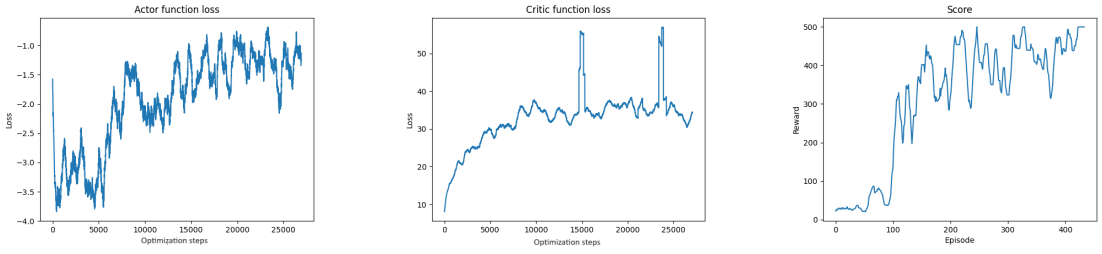
The second variant of A2C we will examine is the *Target networks Advantage Actor-Critic* algorithm, which makes use of target networks. These networks are separate copies of the actor and critic networks that are updated less frequently to provide stable targets during training. To get the best out of our implementation, actor and critic networks are optimized every 5 steps, while target actor and target critic networks converge each 10 steps.

The converging process leverages the *Polyak averaging* algorithm: The *Polyak*

Algorithm 2 Polyak Averaging

Require: τ: Smoothing parameter
Require: *target_network*: Target network
Require: *actual_network*: Actual network
  weights ← Get parameters of *actual_network*
  target_weights ← Get parameters of *target_network*
  for each parameter i in *target_weights* **do**
    $target\_weights[i] \leftarrow weights[i] \cdot \tau + target\_weights[i] \cdot (1 - \tau)$
  end for
  Update *target_network* with *target_weights*

Averaging algorithm is important in the Target Networks A2C variant for making gradual adjustments to the target networks based on the parameters of "explorative" networks. This process ensures smoother training by reducing sudden fluctuations and enhancing overall stability in learning.



(a) Actor loss (b) Critic loss (c) Score

Figure 2.8: A2C with target networks

As we can see from 2.8c the environment was **successfully** solved in 410 episodes requiring around 312 seconds. From figure 2.8b we can see that we have 2 spikes in the Critic Loss, may be due to a data distribution shift caused by a plateau encountered by the actor policy. From figure 2.7c we can see that after only 150

episodes we start getting rewards near the higher bound of 500, and after this it remains quite stable.

2.3.3 Comparison

At a first glance, we notice something unexpected: even if A2C with target networks managed to solve the environment in much less episodes if compared to vanilla A2C **(410 vs 714 episodes)**, the convergence speed is much lower, resulting in **312** seconds for Target Network A2C and **272** secoconds for the Vanilla A2C.

The reason of this is most likely caused by the number of calls to the Polyak Averaging algorithm to converge target networks to actual networks. These calls are made every 10 steps, and in the A2C with targets run around 27k steps were required to solve the environment, leading to a total of 2.7k calls.

On top of that, in our testing environment we had at our disposal only CPUs and RAM. If we had access to GPUs and VRAM, perhaps the convergence time for A2C with target networks could have been reduced, allowing faster access to the model's weights.

2.4 PPO: Proximal Policy Optimization

As we said in the previous paragraphs, *Actor-Critic method* may suffer from instability during training due to the dependency of value estimates on the actor; *vanilla policy gradient methods* can have high variance gradients, which makes training slow and unstable. Furthermore, *trust region policy* optimization (TRPO) is relatively complicated for the quadratic optimization, constraint handling, and the need to calculate the gradient of KL divergence that require advanced understanding of optimization and accurate programming.

In this section we're going to analyze a method characterized by simplicity, stability, computational efficiency, scalability, robustness, and flexibility: the *PPO* algorithm. Its implementation requires less computational effort and time than more complex methods. It is designed to ensure stable and controlled policy updates, reducing the risk of large swings in performance during training. PPO scales well on problems with large state and action spaces and on problems with a large number of parameters to optimize.

The difference between PPO and other RL algorithms is that it uses a trust region constraint to bound the change in the policy. Other algorithms seen before do not use a trust region constraint and may be more prone to instability as a result.

2.4.1 The precursor: TRPO

In 2015, a paper was published introducing the novel TRPO algorithm [2].

TRPO makes use of the Kullback Leibler divergence, which is a distance measure between 2 probability distributions. It makes sure that the new updates policy is not far away from the old policy or, in other words, the new policy is within the trust region of the old policy. It means that policy update won't be deviating largely.

This is the objective function:

$$\underset{\theta}{\text{maximize}} = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t \right] \tag{2.9}$$

subject to

$$\hat{\mathbb{E}}_t[KL[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta \tag{2.10}$$

where θ_{old} is the vector of policy parameters before the update. The theory supporting TRPO actually recommends using a penalty rather than a constraint, effectively solving the problem through unconstrained optimization

$$\underset{\theta}{\text{maximize}} = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t - \beta KL[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)] \right] \tag{2.11}$$

for some coefficient β. This is based on the fact that a specific surrogate objective (which calculates the maximum KL divergence over states rather than the mean) establishes a lower bound (i.e., a pessimistic estimate) on the performance of the policy π.

TRPO opts for a hard constraint instead of a penalty because, selecting a single value for β that works effectively across various problems or even within a single problem as its characteristics evolve during learning, is challenging.

TRPO is a relatively complicated algorithm. The KL constraint adds additional overhead in the form of hard constraints to the optimisation process. We will see the formulation of PPO which is much simpler approach.

2.4.2 PPO: Theoretical formulation

PPO is an improvement on the Trust Region Policy Optimization (TRPO) algorithm. TRPO used a trust region constraint to ensure that the new policy was not too far from the old policy, in order to ensure stability. PPO builds on this idea by using a **clipping** function to bound the change in the policy. This allows PPO to make larger updates to the policy while still ensuring stability. PPO works by iteratively improving the policy through trial and error. The algorithm begins with an initial policy, and then collects data by interacting with the environment and taking actions based on the current policy. This data is used to update the policy in a way that maximizes the expected reward.

The PPO algorithm leverages the advantage of both the Actor-network and the Critic network. The actor tries to maximize the expected return of the policy, and the critic network tries to minimize the error between the estimated return and the actual return.

The advantage value is an important factor in the PPO algorithm because it helps the agent to learn more efficiently by focusing on actions that are more likely to lead to good outcomes.

Next, we have to calculate the loss for actor network. We have to calculate the loss called CPI (Conservative policy iteration). This loss is the expectation ratio between the policy under old parameters to the policy under new parameters multiplied by the advantage value.

The calculated weighted probability value would be larger which results in large parameter updates to the model. To avoid this, PPO adds one more additional parameter called epsilon. With the help of this epsilon, we will clip the ratio between 1- epsilon to 1+epsilon. Finally, we will calculate actor loss by taking the minimum value between the clipped value to the unclipped value multiplied by the advantage.

Instead, the critic loss is calculated as the mean squared error (MSE) between the predicted value estimates and the actual value estimates.

PPO Objective Function

Let

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \tag{2.12}$$

so $r(\theta_{old})$= 1. TRPO maximizes a "surrogate" objective

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t] \tag{2.13}$$

Without adding constraints, this objective function can lead to instability due to excessively large policy updates.

PPO imposes policy ratio, r(θ) to stay within a small interval around 1. That is the interval between $1 - \varepsilon$ and $1 + \varepsilon$. ε is a hyperparameter and in the original PPO paper, it was set to 0.24. This is the objective function of PPO:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t) \right] \tag{2.14}$$

The first term inside the min is L^{CPI}. The second term, clip($r_t(\theta)$, 1$-\varepsilon$, 1$+\varepsilon$)$\hat{A}_t$, the function clip truncates the policy ratio between the range [$1 - \varepsilon$, $1 + \varepsilon$]. Finally, the function clip truncates the policy ratio between the range, so the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective.

Similar to what we discussed in vanilla policy gradient section above, positive advantage function indicates the action taken by the agent is good. On the other hand, a negative advantage indicated bad action. For PPO, in both cases, the clipped operation makes sure it won't deviate largely by clipping the update in the range. Note that L^{CLIP} (θ) = L^{CPI} (θ) to first order around θ_{old} (i.e., where r = 1), however, they become different as θ moves away from (θ_{old}).

Algorithm 3 PPO with Clipped Objective

Input: initial policy parameter θ_0, clipping threshold ϵ
for k=0,1,2, ... **do**
Collect set of partial trajectories D_k on policy $\pi_k = \pi(\theta_k)$
Estimates advantages $\hat{A}_t^{\pi k}$ using any advantage estimation algorithm
Compute policy update

$$\theta_{k+1} = \arg\max_\theta L_{\theta_k}(\theta)^{CLIP}(\theta) \tag{2.15}$$

By taking K steps of minibatch SGD (via Adam), where $L^{CLIP}(\theta)$ = (2.14)
end for

2.4.3 PPO implementation

In the original paper the authors used a neural network architecture that shares parameters between the policy and value function, and therefore they formulated a loss function that combines the policy surrogate, a value function error term and an entropy bonus to ensure sufficient exploration. This is the formula that they had to maximize:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.16}$$

where c1 and c2 are coefficients, S denotes an entropy bonus, and L_t^{VF} is a squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$.

One approach to policy gradient implementation, particularly effective and commonly used with recurrent neural networks, involves running the policy for T timesteps (where T is significantly shorter than the episode length) and using the gathered samples for an update. This method necessitates an advantage estimator that does not consider timesteps beyond T. The estimator employed in this scenario is

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{2.17}$$

where t specifies the time index in [0, T], within a given length-T trajectory segment. Generalizing this choice, we can use a truncated version of generalized advantage estimation, which reduces to equation (2.17) when $\lambda = 1$:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.18}$$

where

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.19}$$

This new family of policy gradient methods for reinforcement learning (PPO) is a reinforcement learning algorithm that is designed to be efficient and stable. It is an on-policy algorithm, which means that it learns from the actions taken within the current policy, rather than from a separate set of data. Whereas standard policy gradient methods perform one gradient update per data sample, this novel objective function enables multiple epochs of minibatch updates.

PPO algorithm uses fixed-length trajectory segments. Each iteration, each of N (parallel) actors collect T timesteps of data. Then we construct the surrogate loss on these NT timesteps of data, and optimize it with minibatch SGD, for K epochs.

Algorithm 4 PPO, Actor-Critic Style

for iteration=1,2, ... **do**
  for actor=1,2,...,N **do**
    Run policy π_{old} in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

2.4.4 Our implementation

Our implementation presents a significant difference with the original paper. This is the use of 2 different neural network, one for the policy (actor) and another one for the value function (critic), and this means that we will compute two different loss functions:

- Actor loss in PPO is based on the *clipped surrogate objective*, which is designed to limit how much the new policy can differ from the old policy. The objective is to maximize this surrogate function, which is expressed as:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{2.20}$$

Where: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio of the probability of the new policy to that of the old policy and ϵ is the clipping parameter.

The term $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ limits the value of $r_t(\theta)$ to prevent too large updates, improving stability of optimization.

- Critic loss (the value function) is calculated using the Mean Squared Error (MSE) between the critic predicted values $V(s_t)$ and the discounted return values R_t. The formula is:

$$L^{VF}(\theta) = \frac{1}{2}\mathbb{E}_t\left[(R_t - V(s_t))^2\right] \tag{2.21}$$

Where: R_t is the accumulated discounted return and $V(s_t)$ is the value estimated by the critic for the state s_t.

When optimizing, the algorithm takes these loss functions and computes gradients with respect to the policy and critic parameters. Then, use an optimizer like Adam to update the neural network parameters.

2.4.5 Results

We conducted an analysis by varying the epsilon parameter across different runs. The results, encapsulated in the plots provided, illustrate the performance trajectories under each configuration of epsilon.

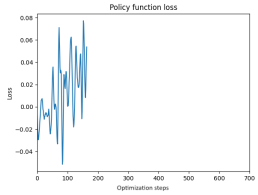
Each row represents a unique combination of the epsilon parameter, ranging from **0.01 to 1**. If ϵ is low we expect the gradient to be quite always clipped, leading to a really slow convergence. On the other hand, if ϵ approaches 1, the ratio between actual π_θ and old $\pi_{\theta_{old}}$ won't be clipped at all, possibly leading to huge changes in the policy between iterations.
These changes should be underlined in the graph by plateaus and discontinuities in the score.
Every graph was **scaled** to make the comparison fair.

Some Observations:

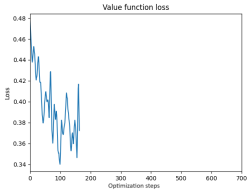
- From 2.9, it is evident that graphs appear truncated. This truncation occurs because, with an epsilon value of 0.01, the policy has an hard time converging. Consequently, fewer steps are performed per episode, resulting in a limited number of optimizations. According to the original paper, optimizations are conducted every 2000 steps; thus, the insufficient number of steps due to the poor convergence with this epsilon value leads to the observed truncation in the plots.

- Using $\epsilon = 0.05$ the situation changes, since the policy appears to be slowly converging and the episode score starts to increase in the final episodes. As we can see in 2.10c, the number of times in which the ratio between old policy and actual policy is clipped is rather high, with around 800k clips over 1200k steps. This means that approximately $\approx 2/3$ of the times the ratio is clipped in the range [0.95, 1.05].

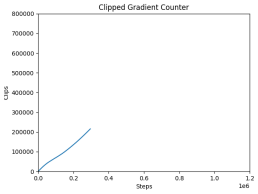
- It's interesting to compare 2.11 and 2.12, since they seem the better performing configurations. Using $\epsilon = 0.2$ makes the policy loss more stable and the episode score approaches 500 with steadiness. On the other hand, using a greater value



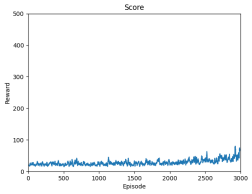
(a) Policy Loss



(b) Critic Loss

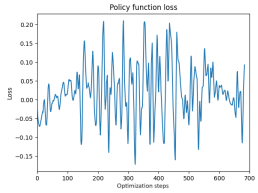


(c) Gradient clips

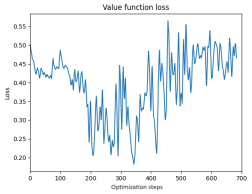


(d) Score

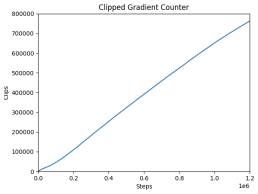
Figure 2.9: $\epsilon = 0.01$



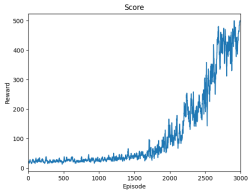
(a) Policy Loss



(b) Critic loss

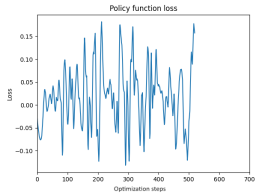


(c) Gradient Clips

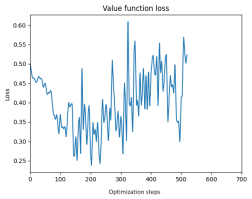


(d) Score

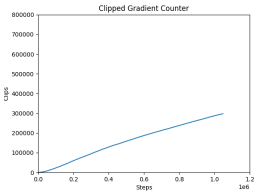
Figure 2.10: $\epsilon = 0.05$



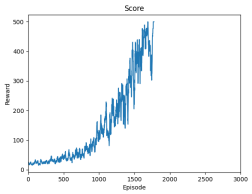
(a) Policy loss



(b) Critic loss

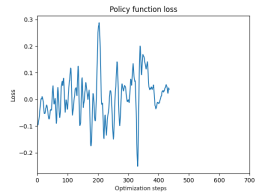


(c) Gradient Clips

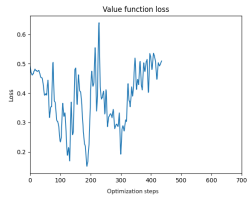


(d) Score

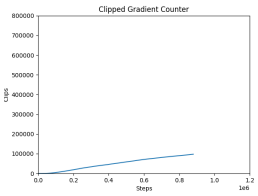
Figure 2.11: $\epsilon = 0.2$



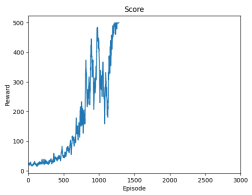
(a) Policy loss



(b) Critic loss

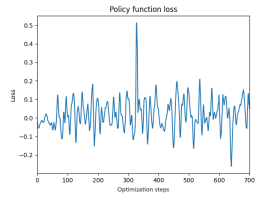


(c) Gradient Clips

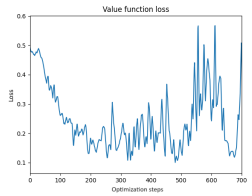


(d) Score

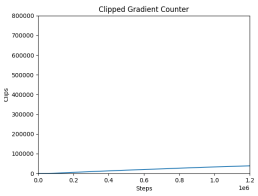
Figure 2.12: $\epsilon = 0.7$



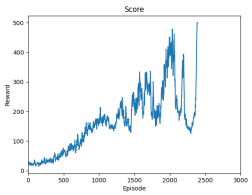
(a) Actor loss



(b) Critic loss



(c) Score



(d) Score

Figure 2.13: $\epsilon = 1$

as $\epsilon = 0.7$ makes the convergence faster but more unstable. It's easy to see from 2.12d that the policy encountered a plateau between episodes 1000 and 1100, leading to a negative spike in the policy loss.

- In the last row 2.13 we have performances with $\epsilon = 1$, which means that the ratio between π_θ and $\pi_{\theta_{old}}$ will be clipped only if lower than 0 or higher than 2 (basically to be clipped the ratio should be more than doubled). From 2.13c we can see that we have around 50k ratio clips over 1200k total optimization steps. This means that approximately $\approx 4\%$ of the times the ratio was above 2 or below 0. One last observation: as we can see we have a lot of plateaus in 2.13d which lead to a great number of episodes to achieve convergence, which anyway may be unstable, due to all the things said above.

Chapter 3

Conclusions

In this final chapter, we delve into a comprehensive examination of the convergence times achieved by various reinforcement learning algorithms in tackling the CartPole-v1 benchmark.

The comparison will be performed only between algorithms that actually managed to solve the environment.

The metric that we've chosen to compare the performances of the different algorithms that we've examined is the **convergence time**, since it's the only truly invariant and independent measure.

We've done several runs and we've taken the average, in order to have some statistical evidence.

3.1 Results

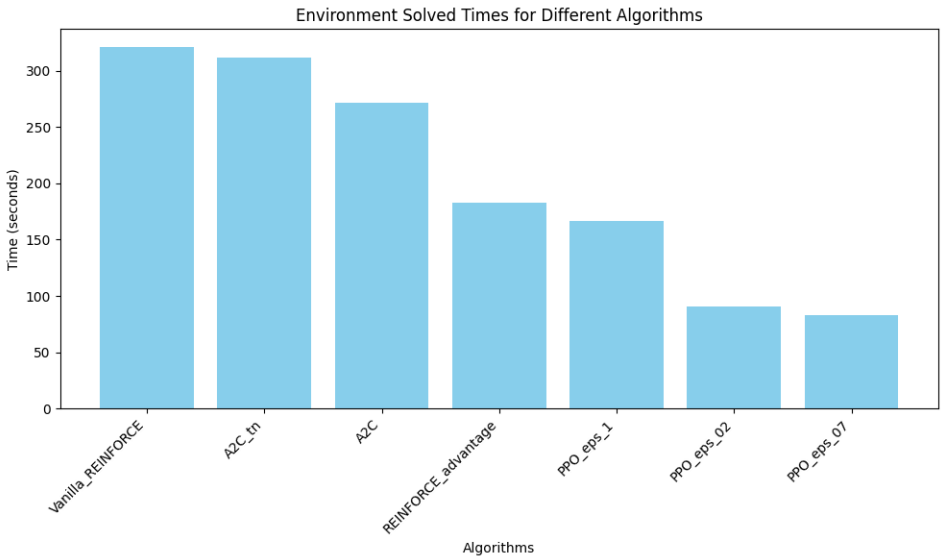


Figure 3.1: Convergence time comparison

The bar chart in 3.1 presents a visual representation of the convergence times for different reinforcement learning algorithms applied to the CartPole-v1 benchmark.

- The **Vanilla REINFORCE** algorithm exhibits the longest convergence time, with 321 seconds. As we've seen from the previous chapter, the usage of the advantage function made the convergence a lot faster, resulting in 183 seconds on average for convergence with **advantage REINFORCE**.

- The **Vanilla A2C** algorithm sits at approximately 272 seconds. Similarly, the **A2C with Target Networks** algorithm shows a convergence time of approximately 312 seconds, which is slower than A2C due to the continuous need to copy weights from actual network to the target network. The number of episodes needed to converge on the other hand is much lower, indicating that if the benchmark is particularly computationally heavy, it may be the better choice over Vanilla A2C. As we can see, REINFORCE with the usage of the advantage function outperforms both A2C and A2C with target networks. One reason for that might be the simplicity of the environment, having a small state and action space and not overly complex dynamics. If we had to deal with a larger state and action space, we could have noticed a higher variance due to the usage of Gt, that could have led to slower convergence.

- Moving on to the **Proximal Policy Optimization** (PPO) algorithms, we have the best performances. We've included in the comparison the 3 best configurations: PPO with eps = 0.7 converges in around 83 seconds, followed closely by PPO with eps=0.2 with a convergence time of 91 seconds. PPO with eps=1 exhibits a slightly longer convergence time, approximately 167 seconds, suggesting that the ratio between older policy and new policy should be ≈ 1

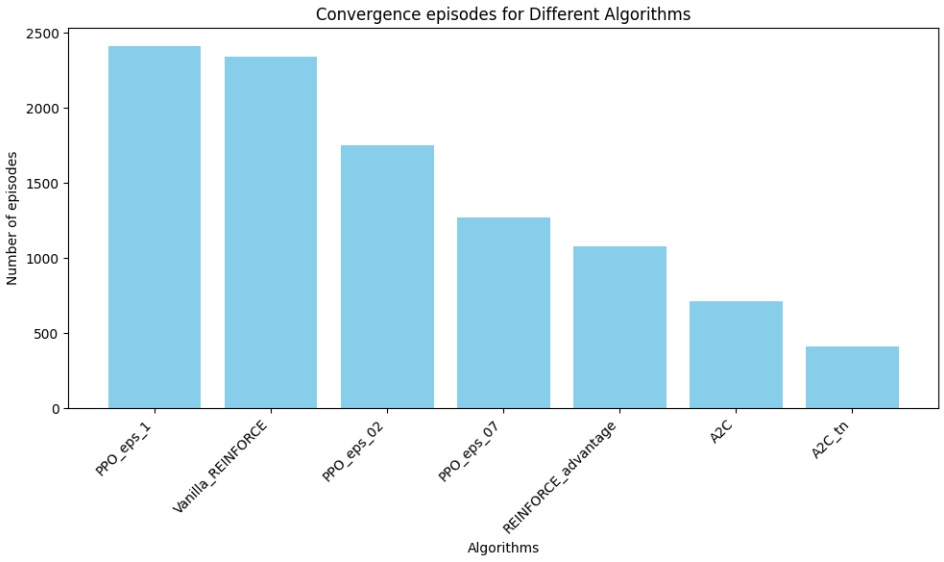
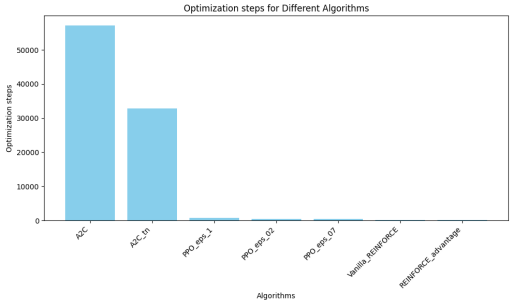
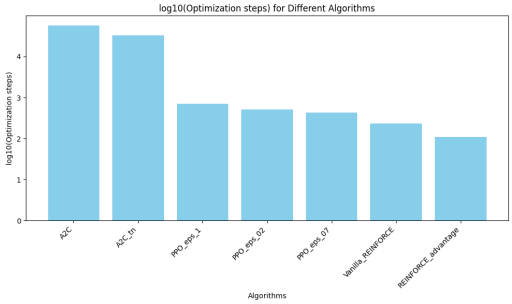


Figure 3.2: Number of episodes to converge comparison



(a) Number of optimization steps to converge



(b) Logarithm of optimization steps to converge

Figure 3.3: Optimization steps to converge

The bar chart in 3.2 presents the **number of episodes required to converge**. This metric can be useful if the environment presents heavy episode that require a lot of time and computational resources to be executed. As we can see, A2C with the usage of target networks is the algorithm that needs the lowest number of episodes to converge, while PPO with ϵ=1 requires around 5 times the number of episodes to reach the early stopping condition.

These consideration are directly related to the **number of optimization steps required to converge**, as shown in 3.3. From 3.3b is quite visible that A2C requires around 2 orders of magnitude more than all the other algorithms. This somehow justifies the fact that A2C algorithms are also the ones requiring the lower number of episodes: for each episode, they perform a huge number of optimization steps if compared to other algorithms. On the other hand, REINFORCE with advantage functions requires the lowest number of optimization steps, but it should be underlined that each step is done on a batch composed of 10 complete trajectories (episodes) and therefore it's huge if compared to the batch size of the other algorithms: 2000 steps for PPO ($\approx$ 5 episodes) and 64 for A2C.

Overall, the comparison highlights the diverse convergence rates among the algorithms, underscoring how **PPO** with its gradient clipping strategy achieves the best time performances, **A2C with target networks** has the lowest number of episodes required to converge, and **REINFORCE with advantage function** has the lowest number of optimization steps required. These aspects should be highlited when chosing one above the other, depending on the environment characteristics.

3.2 Next steps

In light of our comparative analysis of convergence times for various reinforcement learning algorithms on the CartPole-v1 benchmark, several directions for future research may be taken.

An immediate extension would be to test complex algorithms like PPO and A2C not on CartPole-v1 environment, but on more complex and dynamic environments where they need to generalize their learned policies effectively. In fact these algorithms incorporate sophisticated exploration strategies that need to be tested in contexts where simple epsilon-greedy exploration is not sufficient.

An another extension would be to include a detailed comparison with the Asynchronous Advantage Actor-Critic (A3C) and Soft Actor Critic (SAC) algorithms. These model-free reinforcement learning techniques are known for their robust policy optimization capabilities and could offer deeper insights into the efficiency and performance dynamics of different learning strategies.

Furthermore, it would be highly beneficial to explore the deterministic counterparts of the stochastic algorithms evaluated in this study within a continuous action space. This investigation would enable a comprehensive evaluation of the trade-offs between stochastic and deterministic methods in reinforcement learning.

3.3 Source code

Source code in: `https://github.com/git-marco00/reinforcement_learning`

Bibliography

[1] Ivo Grondman et al. "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595.

[2] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.