OVERVIEW - PROXIMAL POLICY OPTIMIZATION ALGORITHMS

REINFORCEMENT LEARNING PROJECT

Gleb Goncharov
M2DS
gleb.goncharov@polytechnique.edu

Marceau Leclerc
M2DS
marceau.leclerc@polytechnique.edu

ABSTRACT

This report is a complementary summary of Proximal Policy Optimization Algorithms [1]. *We will provide the method after having contextualized it and expand on how it improved the field's state of the art. We will discuss its benefits in light of our replications of the experiments.*

Proximal Policy Optimization (PPO) is a family of policy gradient methods that introduces a surrogate objective function. Optimizing it reveals to be more stable than past methods, allowing to operate several epochs per update. This yields a better data-efficiency and model scalability, in addition of being a more robust method by construction of the objective.

The code we produced in the framework of this project is available on this repository.

Contents

1 Introduction

This reports briefly overviews the progress of Gradient Policy Based methods. The main objective of the report is to study Proximal Policy Optimization, provide a summary of the original publication, and compare it with alternative methods. We begin in Section 2 with a few words about PG, TRPO, A2C, ACER, and Cross-Entropy methods. This section mostly holds reminders and therefore can be skipped by the reader. Then in Section 3 we discuss in more details PPO and present results from [1] in Section 4. Section 5 contains our results and thoughts about the process of training the different methods. We compare PPO against A2C, TRPO, and vanilla PG on several baseline tasks from Gymnasium (ex. OpenAI GYM [2]).

Our experiments show that PPO is the most successful method that consistently requires less manipulation to work properly by a large margin. Alternative methods can perform well, but do exhibit the need for a delicate work on their frameworks, oftentimes failing to learn when hyperparameters are not precisely chosen.

2 Context

Policy optimization methods are fundamental to reinforcement learning, where an agent seeks to learn an optimal policy by directly optimizing the parameters of a stochastic policy. These methods are particularly useful in high-dimensional or continuous action spaces where value-based methods may struggle.

2.1 Policy Gradient Methods

Policy gradient methods aim to directly optimize the policy $\pi_\theta(a_t|s_t)$ by estimating the gradient of the expected return with respect to the policy parameters θ and performing stochastic gradient ascent. The fundamental policy gradient estimator is given by:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{1}$$

where:

- $\pi_\theta(a_t|s_t)$ is a stochastic policy parameterized by θ.
- $\hat{A}_t$ represents an estimator of the advantage function at timestep t, which quantifies the benefit of taking action a_t in state s_t compared to the expected value of the policy.
- $\hat{\mathbb{E}}_t[\cdot]$ represents an empirical expectation over a batch of sampled trajectories.

Implementations of policy gradient methods typically rely on automatic differentiation frameworks, where the objective function whose gradient is computed corresponds to:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right]. \tag{2}$$

Indeed the gradient of L^{PG} with respect to θ is immediately retrieved using the linearity of the expectation :

$$\nabla_\theta L^{PG}(\theta) = \nabla_\theta \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] = \hat{g}. \tag{3}$$

While it it is possible in practice to perform multiple gradient updates using the same trajectory data to improve sample efficiency, empirical results indicate that this often leads to excessively large policy updates, resulting in unstable learning behavior. This is a problem that PPO directly solves as we will see in Section 3.

2.2 Trust Region Methods

Trust Region Policy Optimization (TRPO) [3] addresses instability in policy updates by enforcing constraints on the extent of policy changes between updates. TRPO optimizes a surrogate objective while ensuring that the updated policy remains within a bounded divergence from the previous policy:

$$\max_\theta \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \tag{4}$$

subject to the constraint:

$$\hat{\mathbb{E}}_t \left[KL(\pi_{\theta_{\text{old}}}(\cdot|s_t) || \pi_\theta(\cdot|s_t)) \right] \leq \delta. \tag{5}$$

With KL denoting the Kullback-Leibler divergence :

$$KL(\pi_{\theta_{\text{old}}}(\cdot|s_t)||\pi_\theta(\cdot|s_t)) = \sum_a \pi_{\theta_{\text{old}}}(a|s_t) \log \frac{\pi_{\theta_{\text{old}}}(a|s_t)}{\pi_\theta(a|s_t)}. \tag{6}$$

Here θ_{old} denotes the policy parameters before the update. The constraint ensures that the KL divergence between the old and new policies does not exceed a predefined threshold δ, preventing abrupt changes that could destabilize learning. It is a fairly intuitive and natural counteract to the problem described in Section 2.1.

The constrained optimization problem in TRPO is solved approximately using the conjugate gradient (CG) algorithm, which efficiently computes the step direction for updating the policy. The key problem involves solving a linear system of the form:

$$Ax = b, \tag{7}$$

where A represents the Fisher information matrix (or an approximation of the Hessian of the KL divergence), and b is the policy gradient. Since direct inversion of A is computationally expensive, the conjugate gradient algorithm iteratively finds an approximate solution by minimizing the quadratic form:

$$\frac{1}{2}x^T Ax - x^T b. \tag{8}$$

The iterative update rule is given by:

$$x_{k+1} = x_k + \alpha_k p_k, \tag{9}$$

where p_k is the search direction and α_k is the step size. The search direction is updated using:

$$p_{k+1} = r_{k+1} + \gamma_k p_k, \tag{10}$$

where $r_k = b - Ax_k$ is the residual and γ_k is a scaling factor ensuring conjugacy. The algorithm terminates when the residual norm $\|r_k\|$ falls below a threshold.

This method efficiently finds the step direction for policy updates while avoiding explicit matrix inversion, making it suitable for high-dimensional reinforcement learning problems.

Although the theoretical foundation of TRPO suggests that a penalty formulation could be used instead of a hard constraint, an unconstrained optimization approach introduces difficulties in choosing an appropriate penalty coefficient β that generalizes across different environments and learning stages. Specifically, the penalized objective takes the form:

$$\max_\theta \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t - \beta KL(\pi_{\theta_{\text{old}}}(\cdot|s_t)||\pi_\theta(\cdot|s_t)) \right]. \tag{11}$$

However, fixing a single penalty coefficient β often results in suboptimal performance because the optimal value of β varies dynamically as learning progresses. Empirical studies have shown that simply using a fixed penalty coefficient in stochastic gradient descent (SGD) optimization is insufficient to replicate TRPO's guaranteed monotonic improvement properties. As a result, TRPO employs a hard constraint rather than a penalty to ensure stable learning behavior and reliable policy improvement. Algorithm 1 offers a compact overview of the implementation.

2.3 Advantage Actor-Critic

Advantage Actor-Critic (A2C) is an actor-critic algorithm that bridges the gap between the simplicity of Vanilla Policy Gradients and the stability of Trust Region methods. It combines a value-based critic with a policy-based actor to achieve more stable and efficient learning. It is the synchronous variant of the A3C [4] method.

A2C maintains two function approximators:

- **Actor:** The policy $\pi_\theta(a_t|s_t)$, which selects actions based on the current state.

Algorithm 1 Trust Region Policy Optimization (TRPO)

Initialize policy parameters θ and value function parameters ϕ
for iteration $= 1, 2, \ldots$ **do**
    Collect a set of trajectories $\{\tau_i\}$ by running π_θ in the environment
    Compute advantage estimates $\hat{A}_t$ using the value function V_ϕ
    Compute the policy gradient $\nabla_\theta L(\theta)$ for the surrogate objective
    Use the conjugate gradient (CG) method to approximate the Fisher vector product and find the update direction
    Perform a line search to ensure the KL divergence constraint is satisfied
    Update $\theta \leftarrow \theta_{\text{new}}$ using the accepted step
    Update the value function parameters ϕ (e.g., by minimizing mean squared error)
end for

- **Critic:** The value function $V_\phi(s_t)$, which estimates the expected return from state s_t.

The key innovation in A2C is the use of an advantage estimator to reduce the variance of the policy gradient. The advantage is computed as:

$$\hat{A}_t = R_t - V_\phi(s_t), \tag{12}$$

where R_t is the cumulative return from timestep t. The policy is then updated by maximizing the objective:

$$L^{A2C}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right], \tag{13}$$

with its gradient given by:

$$\nabla_\theta L^{A2C}(\theta) = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]. \tag{14}$$

A2C typically gathers experience from multiple parallel environments and performs synchronous updates. This approach stabilizes learning by reducing variance in the gradient estimates compared to asynchronous methods such as A3C [4]. Additionally, an entropy bonus is often incorporated to promote exploration:

$$L^{A2C}_{\text{total}}(\theta) = L^{A2C}(\theta) + \alpha \hat{\mathbb{E}}_t \left[\mathcal{H}(\pi_\theta(\cdot|s_t)) \right], \tag{15}$$

where $\mathcal{H}(\pi_\theta(\cdot|s_t))$ denotes the entropy of the policy and α is a scaling factor controlling the strength of the regularization.

This balance between reducing variance (through the advantage function) and ensuring robust exploration makes A2C a powerful algorithm for tackling high-dimensional or continuous action spaces. The implementation is detailed in Algorithm 2.

Algorithm 2 Advantage Actor-Critic (A2C)

Initialize policy parameters θ and value function parameters ϕ
for iteration $= 1, 2, \ldots$ **do**
    Run M parallel environments for T timesteps using π_θ
    For each environment, collect transitions (s_t, a_t, r_t, s_{t+1})
    Compute returns R_t and advantage estimates $\hat{A}_t = R_t - V_\phi(s_t)$
    Update θ by ascending the policy gradient:

$$\nabla_\theta \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t)\, \hat{A}_t \right]$$

    Update ϕ by minimizing the value loss (e.g., mean squared error)
    Optionally, add an entropy bonus to encourage exploration
end for

2.4 ACER

Actor-Critic with Experience Replay (ACER) [5] is an off-policy variant of A3C [4] that combines a replay buffer, truncated importance sampling, and trust-region style updates to improve sample efficiency and stability. Unlike purely on-policy methods (e.g., A2C/A3C), ACER can leverage past experiences stored in a buffer, enabling better reuse of trajectories and faster convergence.

A key component of ACER is the use of *truncated importance sampling* to correct for discrepancies between the policy used to generate data and the current policy. By truncating large importance weights, ACER prevents excessively large updates that could destabilize learning. Additionally, a trust region–like constraint is imposed to further bound the policy updates and maintain training stability.

Algorithm 3 outlines the core steps of ACER:

Algorithm 3 Actor-Critic with Experience Replay (ACER)

Initialize policy parameters θ, value function parameters ϕ, and replay buffer $\mathcal{D}$
for iteration $= 1, 2, \ldots$ **do**
  for actor $= 1, 2, \ldots, M$ **do**
    Run policy π_θ in the environment for T timesteps
    Store collected transitions (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  end for
  for update $= 1, 2, \ldots, K$ **do**
    Sample a mini-batch of transitions from $\mathcal{D}$
    Compute truncated importance weights

$$w_t = \min\Big(\rho_t, \rho_{\max}\Big), \quad \rho_t = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)},$$

    where $\rho_{\max}$ is a clipping threshold
    Update policy parameters θ using the off-policy actor-critic loss (with truncated importance sampling and trust-region constraint)
    Update value parameters ϕ by minimizing the critic loss (e.g., mean squared error between $V_\phi(s_t)$ and $r_t + \gamma V_\phi(s_{t+1})$)
  end for
end for

In summary, ACER preserves the benefits of actor-critic algorithms (i.e., low-variance gradient estimates via a learned critic) while incorporating an off-policy replay buffer and stabilizing mechanisms (truncated importance sampling and a trust-region approach). This design yields improved sample efficiency compared to purely on-policy methods.

2.5 Cross-Entropy Method

The Cross-Entropy Method (CEM) [6] is a population-based optimization technique originally introduced for rare-event simulation, but it has been successfully adapted to reinforcement learning for policy search. Unlike gradient-based approaches, CEM does not require the objective to be differentiable. Instead, it samples a population of candidate solutions from a parameterized distribution, evaluates each candidate's performance, and then updates the distribution parameters to focus on the top-performing (or "elite") solutions.

CEM starts by initializing a probability distribution over policy parameters, often a Gaussian with some mean μ_0 and covariance Σ_0. In each iteration, it samples a set of policy parameters from this distribution and rolls out the corresponding policies in the environment to compute their returns. The best-performing candidates, usually a fixed number or a certain top percentile, are then used to update the mean and covariance of the distribution. This process iterates until a desired performance threshold is met or until convergence is observed. A pseudocode is provided in Algorithm 4.

A drawback of the method is that it often requires a large number of samples to converge, especially in high-dimensional spaces, which can be computationally expensive. Despite these limitations, its simplicity and effectiveness make it a popular baseline for policy search.

3 Method

Proximal Policy Optimization is a family of algorithms that aims to improve policy gradient methods. Like TRPO but through a different strategy limited to the first order, PPO hinders brutal policy updates. We will in this section detail the method and its algorithm.

Algorithm 4 Cross-Entropy Method (CEM) for Policy Search

Input: Number of samples N, number (or fraction) of elites K, initial distribution parameters (μ_0, Σ_0)
Initialize: $\mu \leftarrow \mu_0,\ \Sigma \leftarrow \Sigma_0$
for iteration $= 1, 2, \ldots$ **do**
    Sample $\{\theta_i\}_{i=1}^{N}$ from $\mathcal{N}(\mu, \Sigma)$
    for each θ_i **do**
        Evaluate return $J(\theta_i)$ in the environment
    end for
    Identify top K candidates $\{\theta_k\}_{k=1}^{K}$ based on $J(\theta_i)$
    Update $\mu \leftarrow \frac{1}{K}\sum_{k=1}^{K}\theta_k$
    Update $\Sigma \leftarrow \frac{1}{K}\sum_{k=1}^{K}(\theta_k - \mu)(\theta_k - \mu)^T$
    if termination criterion met (e.g., $J(\theta)$ exceeds threshold) **then**
        break
    end if
end for
Output: Learned distribution parameters (μ, Σ) or best candidate policy

3.1 Theoretical Framework

The key idea behind PPO is to control the update step to ensure that the new policy remains close to the old one. Given a policy parameterized by θ, we define the probability ratio

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

which compares the likelihood of an action under the new policy against the old policy. A straightforward surrogate objective, derived from conservative policy iteration, is

$$L^{\text{CPI}}(\theta) = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right],$$

where $\hat{A}_t$ denotes an estimator of the advantage function. Although maximizing $L^{\text{CPI}}(\theta)$ would improve performance, it may also lead to overly aggressive policy updates.

To address this, PPO introduces a clipped surrogate objective:

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

with ϵ as a hyperparameter (typically set to 0.2). The clipping function restricts the probability ratio to the interval $[1-\epsilon,\, 1+\epsilon]$, which prevents the update from deviating too far from the current policy. Specifically, when the advantage is positive, any increase in $r_t(\theta)$ beyond $1+\epsilon$ is curtailed; similarly, for negative advantages, decreases below $1-\epsilon$ are limited.

This formulation ensures that near θ_{old} (where $r_t(\theta) \approx 1$), the clipped objective closely approximates the original surrogate. However, as the policy diverges, the clipping mechanism provides a conservative correction, effectively taking a pessimistic bound on the improvement. In this way, PPO maintains a balance between policy improvement and stability, reducing the risk of destabilizing updates during training. Figure 1 provides a visualization of L^{CLIP}

Because L^{CLIP} is constructed for stability, it is reasonable to operate several updates of $\pi_\theta(a_t|s_t)$ using a single batch of timesteps. All in all, this makes PPO pretty data-efficient.

An interesting sidenote is that A2C can be seen as a particular case of PPO. Indeed if we set the number of epochs per update to 1 and have the same data, [7] shows the algorithms are equivalent ot one another. However, this hinders one of the most important upsides of PPO.

3.2 Alternative formulation : Adaptive KL Penalty

PPO's logic can be alternatively formulated under a transportation objective. It simply involves dynamically changing the KL penalty coefficient β as detailed in Algorithm 5. [1] uses it as a baseline, and shows it is inferior to the "straight-forward" PPO method that we presented in Section 3.1 and for which we will discuss algorithms in Section 3.3.

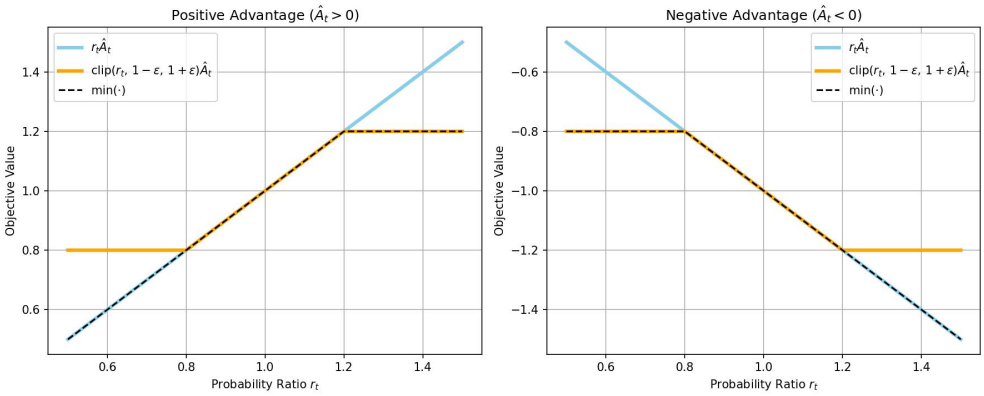

Figure 1: Visualization of L^{CLIP} (dashed line).

Algorithm 5 KL-Penalized Objective Optimization

Input: Policy parameters θ, old policy parameters θ_{old}, target KL divergence d_{targ}, initial penalty coefficient β
for each epoch of minibatch SGD **do**
    Optimize the following objective:

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t - \beta\,\mathrm{KL}\Big(\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t)\Big)\right]$$

end for
Compute

$$d = \hat{\mathbb{E}}_t\left[\mathrm{KL}\Big(\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t)\Big)\right]$$

if $d < \dfrac{d_{\text{targ}}}{1.5}$ **then**
    $\beta \leftarrow \beta/2$
else if $d > 1.5\, d_{\text{targ}}$ **then**
    $\beta \leftarrow \beta \times 2$
end if

3.3 Main Algorithms

A typical "vanilla policy gradient" (VPG) algorithm can be modified easily to integrate the loss of interest, either L^{CLIP} or L^{KLPEN}. A pseudocode is available in Algorithm 6

Algorithm 6 Typical Policy Gradient Implementation

Initialize policy parameters θ
while not converged **do**
    Collect a batch of trajectories (or episodes) $\{(s_t, a_t, r_t)\}$ by rolling out the current policy π_θ
    Compute the chosen policy gradient loss. For example:

- Vanilla PG: $L^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_\theta(a_t \mid s_t)\, \hat{A}_t\right]$
- Clipped PPO: $L^{CLIP}(\theta)$
- KL-penalized: $L^{KLPEN}(\theta)$

    Use automatic differentiation to compute $\nabla_\theta L(\theta)$
    Update the policy parameters $\theta \leftarrow \theta + \alpha\, \nabla_\theta L(\theta)$
end while

In practice and as per in [4] and [8], we adopt the following approach:

- **Variance-reduced Advantage Function:** Most methods involve learning a state-value function $V(s)$ (e.g., generalized advantage estimation [8] or the finite-horizon estimators in [4]).
- **Combined Loss Function:** Typically, as the policy and value function share parameters, one adds two terms to the L^{CLIP} loss as described below the next objective :

$$L_{\text{CLIP}+VF+S}(\theta) \;=\; \mathbb{E}_t\Big[\, L_{\text{CLIP}}(\theta) \;-\; c_1\, L_V^F(\theta) \;+\; c_2\, S[\pi_\theta](s_t)\Big],$$

  which is (approximately) maximized each iteration. Here:
  - $L_{\text{CLIP}}(\theta)$ is the clipped surrogate objective.
  - $L_V^F(\theta)$ is a squared-error loss, here to help accurately predict expected returns,

$$L_V^F(\theta) \;=\; \big(V_\theta(s_t) - V_t^{\text{targ}}\big)^2.$$

  - $S[\pi_\theta](s_t)$ is an entropy bonus to ensure sufficient exploration. Typically :

$$S[\pi_\theta](s_t) \;=\; \mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}\big[-\log \pi_\theta(a_t \mid s_t)\big]. \tag{16}$$

  - c_1, c_2 are scalar coefficients that balance the value loss and the entropy term, respectively.
- **Truncated Rollouts (Actor-Critic Style):** For an update, we run the policy for T timesteps (with $T \ll$ episode length). An appropriate (truncated) advantage estimator as presented in [4] is :

$$\hat{A}_t \;=\; -V(s_t) \;+\; r_t \;+\; \gamma\, r_{t+1} \;+\; \ldots \;+\; \gamma^{\,T-t+1}\, r_{T-1} \;+\; \gamma^{\,T-t}\, V(s_T),$$

  where $t \in [0, T]$ indexes the steps within each length-T trajectory segment.

Algorithm 7 PPO, Actor-Critic Style

for iteration = 1, 2, … **do**
  for actor = 1, 2, …, N **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in the environment for T timesteps
    Store the collected transitions (s_t, a_t, r_t, s_{t+1}) for $t = 1, \ldots, T$
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$ (e.g., using GAE [8])
  end for
  Construct the surrogate loss (e.g., L_{CLIP}) on these $N \times T$ samples
  Optimize the surrogate loss w.r.t. θ using minibatch SGD or Adam, for K epochs
  $\theta_{\text{old}} \leftarrow \theta$ *(Update old policy parameters)*
end for

4 Experiments in [1]

Here, we will provide a digest of the main take-aways from the original PPO article [1]. Important tables and figures are taken from it directly.

4.1 Ablation studies

Focusing on the OpenAI Gym [2] and using the MuJoCo engine [9], a fair comparison can be made to determine which objective - no clipping, L^{CLIP} and L^{KLPEN}, adaptive or not - performs the best in practice. Each algorithm was run 3 times for all environments over 1M timesteps. Results were scaled and averaged. Table 1 offers the full comparison, clearly showing that the clipped surrogate objective is the best alternative.

Another important remark to draw from Table 1 is that the KL penalty algorithm performs similarly with or without adaptive coefficient β. Even though it doesn't score as high as L^{CLIP} with $\epsilon = 0.2$, it still performs much better than the VPG algorithm.

4.2 Results

The combination of OpenAi Gym [2] and MuJoCo [9] environments is a well known benchmark used to compare various methods on an even ground. The benchmark used at the time of publication was relying on the "-v1"'s of the various games. A detailed description that includes the observation and action spaces of the "-v4" versions is provided in Table 3, part of Section 5.

PPO with parameters selected as per presented in Section 4.1 was compared on this benchmark against:

algorithm	avg. normalized score
No clipping or penalty	-0.39
Clipping, $\epsilon = 0.1$	0.76
Clipping, $\epsilon = 0.2$	**0.82**
Clipping, $\epsilon = 0.3$	0.70
Adaptive KL $d_{\text{targ}} = 0.003$	0.68
Adaptive KL $d_{\text{targ}} = 0.01$	0.74
Adaptive KL $d_{\text{targ}} = 0.03$	0.71
Fixed KL, $\beta = 0.3$	0.62
Fixed KL, $\beta = 1$	0.71
Fixed KL, $\beta = 3$	0.72
Fixed KL, $\beta = 10$	0.69

Table 1: Comparison of PPO variants and their average normalized scores.

- A2C [4], see Section 2.3, with and without trust region [5] to reduce the size of the policy update, in the same philosophy as TRPO.
- The CEM[6], see 2.5.
- Vanilla PG (Section 2.1) with an adaptive stepsize.
- TRPO [3], see Section 2.2.

The procedures were run over 1M timesteps, and plotting the returns as in Fig. 2 show both the end quality of learned policies as well as how fast they were learned.

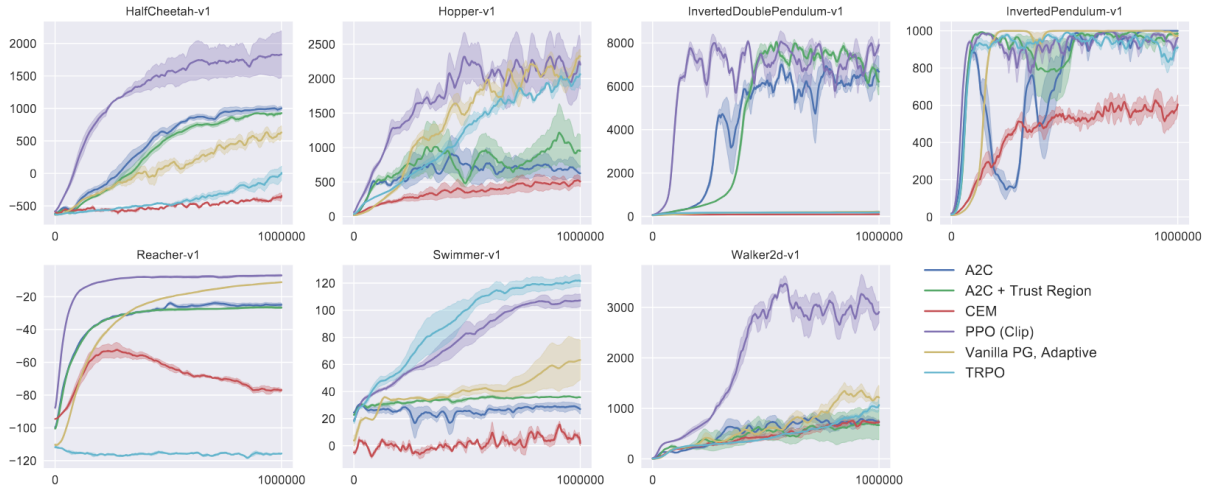

Figure 2: OpenAI Gym benchmarks from [1].

Fig. 2 clearly shows that PPO outclasses all other mentioned methods both in terms of quickness to learn as well as quality of final policies, in all but one (Swimmer-v1) environment, where it is slightly but noticeably outperformed by TRPO.

PPO was also measured against ACER [5] (outlined in Section 2.4) and A2C on the 49 Atari games included in OpenAI Gym [2] at time of the publication.

Results displayed in Table 2 were that PPO learned noticeably faster than ACER - having the better average reward over the whole training - but did have lower reward on the last 100 episodes more often than not. A2C was outclassed in all but one scenario.

	A2C	ACER	PPO	Tie
(1) avg. episode reward over all training	1	18	**30**	0
(2) avg. episode reward over last 100 episodes	1	**28**	19	1

Table 2: Comparison of average episode rewards for A2C, ACER, and PPO. "Tie" indicates the number of tasks on which algorithms performed equally.

5 Experimental Reproduction

5.1 Framework

We aimed to implement PPO and alternative methods in order to reproduce experiments in Section 4.2. In particular, reproducing some of the trainings on the OpenAI [2] benchmarks (see Fig. 2). We added to these 7 environments "HumanoidStandup-v4" which we deemed interesting for its high dimensional observation and action spaces. A detailed overview of all 8 environment and their complexity is available in Table 3.

Environment	**Obs. Dim.**	**Act. Dim.**	**Max Steps**	**Notes**
HalfCheetah-v4	17	6	1000	Forward locomotion task
Hopper-v4	11	3	1000	Single-legged hopping
InvertedDoublePendulum-v4	11	1	1000	Balance a two-link pendulum
InvertedPendulum-v4	4	1	1000	Classic single-pendulum balance
Reacher-v4	11	2	50	2D reaching task
Swimmer-v4	8	2	1000	Underwater forward swimming
Walker2d-v4	17	6	1000	Biped locomotion
HumanoidStandup-v4	376	17	1000	Stand up from flat pose

Table 3: Key characteristics of selected MuJoCo Gym environments. *Observation Dim.* and *Action Dim.* refer to the sizes of the state and action vectors, respectively. *Max Steps* is the default episode length limit.

For each of the method and environment pairs, we conducted 3 experiments in order to get a better grasp of the general trends. PPO was implemented closely following the original paper and Section 4.1. The most relevant hyperparameters chosen for these experiments are available in Table 4.

Hyperparameter	**Value**
Number of Cells (per layer)	256
Starting Learning Rate	3×10^{-4}
Maximum Gradient Norm	1.0
Frames per Batch	1000
Sub-Batch Size	64
Number of Epochs (per batch)	10
Clipping threshold (ϵ in L^{CLIP})	0.2
Discount Factor (γ)	0.99
GAE Lambda (λ)	0.95
Entropy Epsilon	10^{-4}

Table 4: PPO Hyperparameters

The other methods we implemented were TRPO ([3], Section 2.2), VPG (Section 2.1) and A2C ([4], Section 2.3). We will now elaborate on what was observed.

5.2 Results

Our results show that PPO strongly outperforms all of the other methods. It manages to learn faster and get a higher score after 1M timesteps. A2C was the second best method. It managed to learn somewhat consistently but its results

were significantly worse than the ones from PPO. The other two methods required way more careful parameters tuning and their results were not promising during experiments. However, if tuned properly these two methods can also be working solutions for these environments. Our results are shown on Fig. 3. The testing time rewards for episodes are plotted there.

It has to be said that for PPO, we had a much better idea of the starting points (i.e. sets of hyperparameters) that would work in the context of these experiments. Even though we did find efficient set-ups for A2C rather quickly, we never quite found parameters that made TRPO and VPG work. This is why in our results these two methods seem almost useless, but as show in Section 4.2, there clearly is way to make these methods perform.

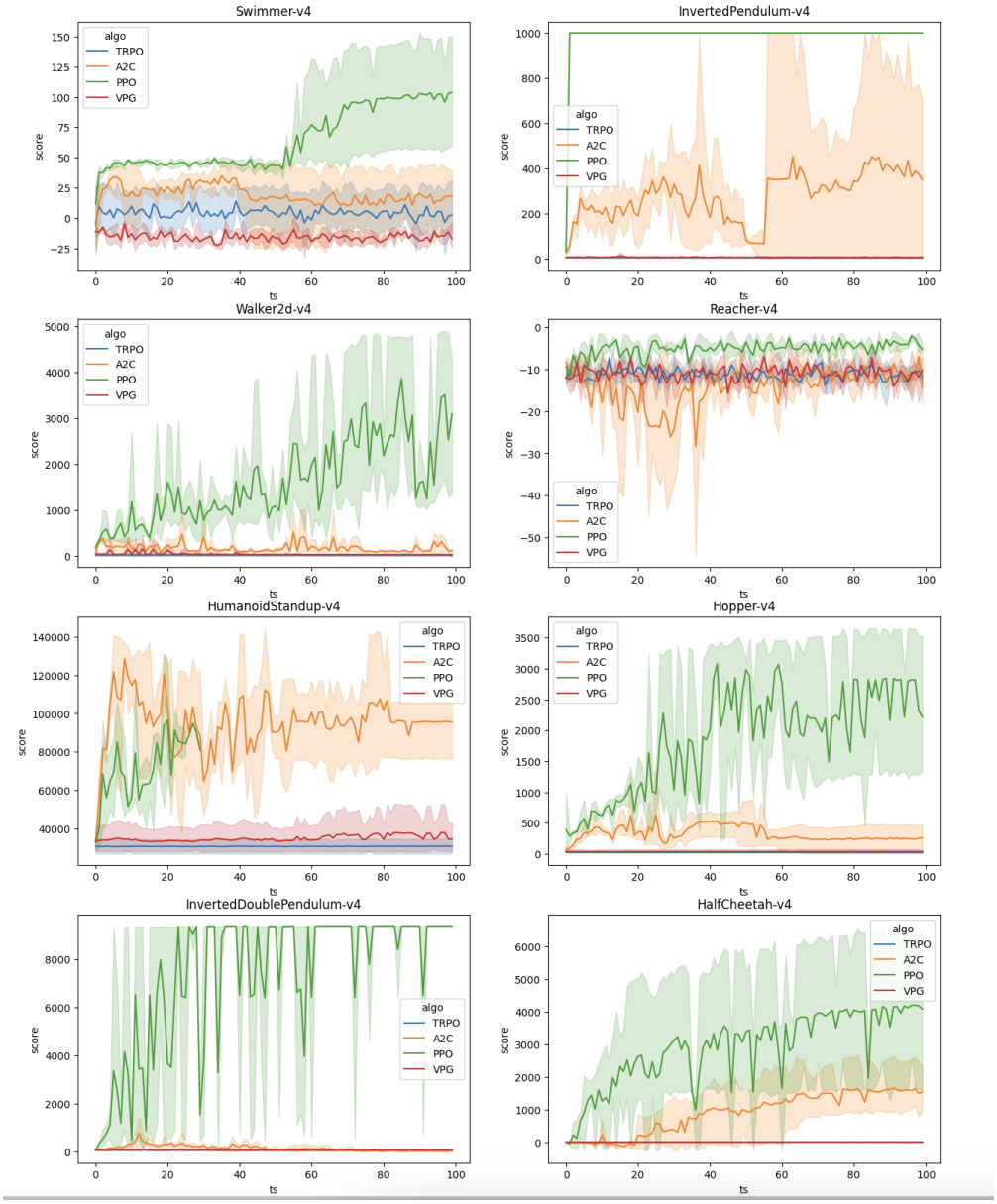

Figure 3: Results of our reproductions.

Even though PPO did significantly better, we still observed that what was learned was typically a single move to "throw" in some way the subject towards an important reward. For instance, in Fig. 4, we can see that the PPO algorithm just learned a single leap forward, which ended up making it the nest of all four procedures.

A last observation pertains to the HumanoidStandup-v4 environment. With PPO, all 6 runs we tried ended up printing "simulation unstable" warnings, as a result of a joint reaching infinite values. We cannot explain this, as PPO by design doesn't allow for brutal changes in the policy. We still kept the plot, as it shows PPO learns slower than A2C for roughly the first 300k frames but does match its performance at that point.

6 Discussion

Our study highlights the robustness and efficiency of Proximal Policy Optimization (PPO) compared to other policy gradient methods. The experiments and ablation studies presented earlier show that PPO consistently learns faster and achieves higher rewards across various benchmark environments, confirming its design advantages.

The clipped surrogate objective, which limits the policy update step, plays a crucial role in ensuring stability during training. This mechanism not only prevents overly aggressive updates but also allows multiple epochs per update, leading to better data efficiency. In contrast, alternative methods such as TRPO and vanilla policy gradients demand more careful hyperparameter tuning and can be less robust in practice. Although A2C performs competitively, it still falls short of the overall performance and reliability offered by PPO.

Moreover, the reproduction experiments emphasize the practical benefits of PPO's architecture. The ability to reuse trajectories effectively and the inherent simplicity in its implementation contribute to its widespread applicability, especially in high-dimensional and continuous control tasks.

In summary, the results reinforce the notion that PPO's balance between exploration, exploitation, and stability makes it a compelling choice for reinforcement learning applications. Future work could explore further refinements in hyperparameter optimization for competing methods or hybrid approaches that incorporate the best elements from each algorithm.

References

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

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[3] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2017.

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

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

[6] I. Szita and A. Lörincz, "Learning tetris using the noisy cross-entropy method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, 2006.

[7] S. Huang, A. Kanervisto, A. Raffin, W. Wang, S. Ontañón, and R. F. J. Dossa, "A2c is a special case of ppo," 2022.

[8] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[9] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

A Additional Material

Figure 4 shows the grid of screenshots from the Walker2d-v4 environment using PPO.

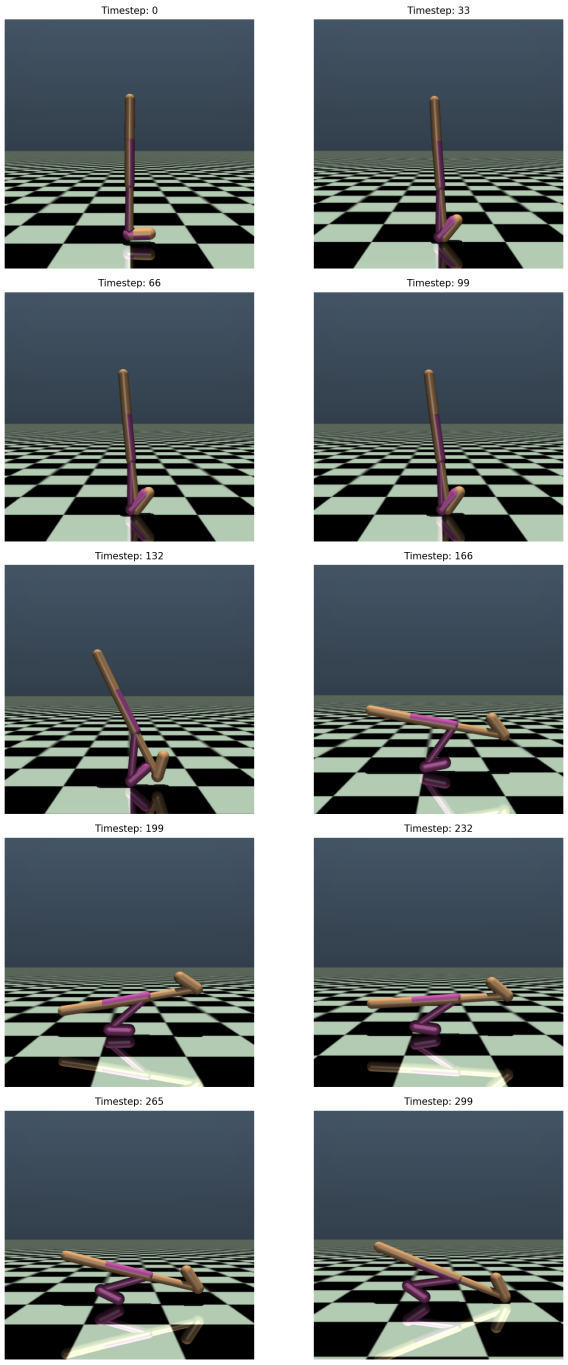

Figure 4: Grid of screenshots from the Walker2d environment using PPO.

Figure 5 shows the grid of screenshots from the Walker2d-v4 environment using A2C.

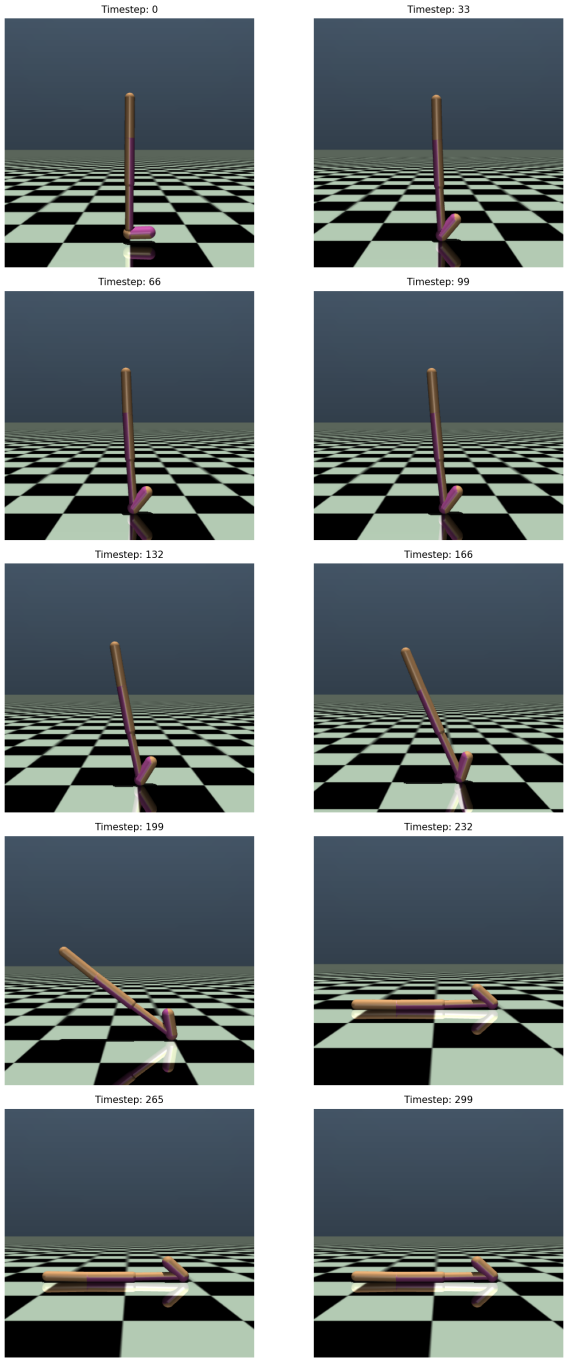

Figure 5: Grid of screenshots from the Walker2d environment using A2C.

Figure 6 shows the grid of screenshots from the Swimmer-v4 environment using PPO.

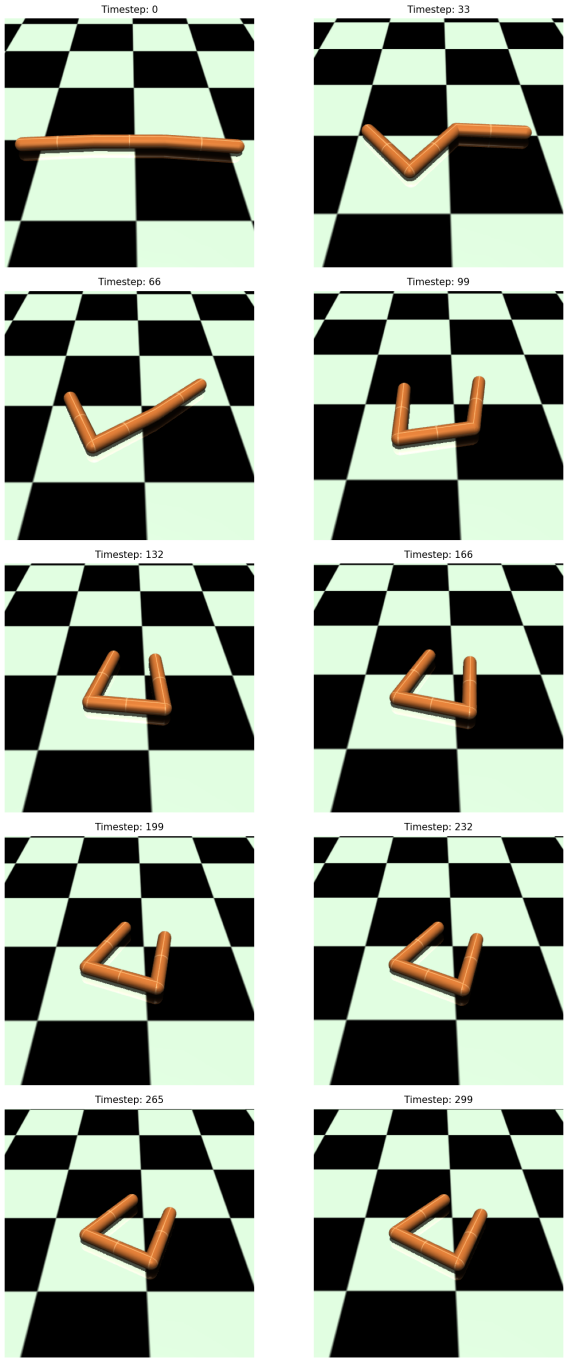

Figure 6: Grid of screenshots from the Swimmer environment using PPO.

Figure 7 shows the grid of screenshots from the Swimmer-v4 environment using A2C.

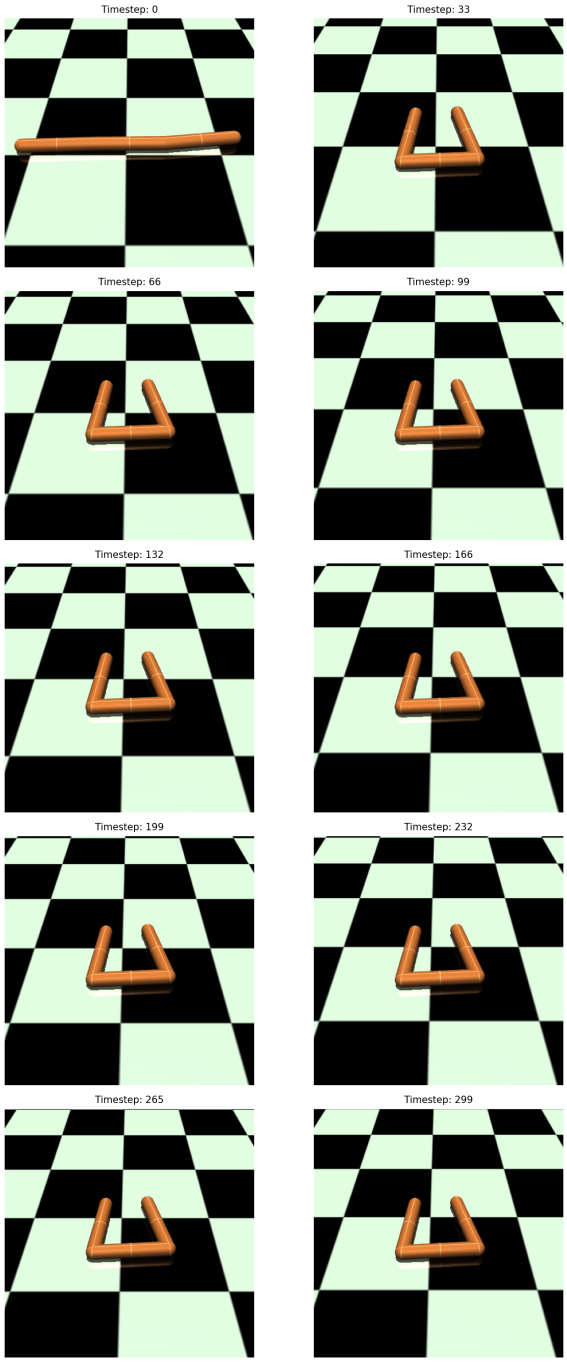

Figure 7: Grid of screenshots from the Swimmer environment using A2C.