REINFORCE vs Actor-Critic

Both are policy gradient algorithms

**REINFORCE**

It optimizes the expected return by following the gradient of the log-policy scaled by the returns obtained from full episodes.

Algorithm summary:

-Collect full episodes using the correct policy (Monte Carlo)

-Compute return

-Update parameters with

-Update parameters with baseline

The baseline is used to reduce the variance, it can be either a static value (b=20) or estimated as the value function (like actor-critic).

The loss is computed by adding an entropy coefficient which encourages exploration.

Characteristics:

-Simple and unbiased gradient estimate

-No value function required

-High variance due to full-episode returns

-Inefficient sample usage (one update per episode)

-Convergence can be very slow

**Actor-Critic**

It combines REINFORCE and value-based methods. It uses two models:

Actor: learns the policy

Critic: learns a values function to estimate expected returns

It uses temporal-difference learning to estimate the advantage

In the updated version the advantages are computed with General Advantage Estimation (GAE) to balance bias-variance via λ.

Update rule

To stabilize learning multiple updates are used in the critc

Characteristics:

-Lower variance due to baseline V(s)

-Online updates possible

-Faster and more stable learning

-Biased gradient estimate due to bootstrapping

-Additional model must be trained (extra complexity)

-Sensitive to the quality of the value estimate

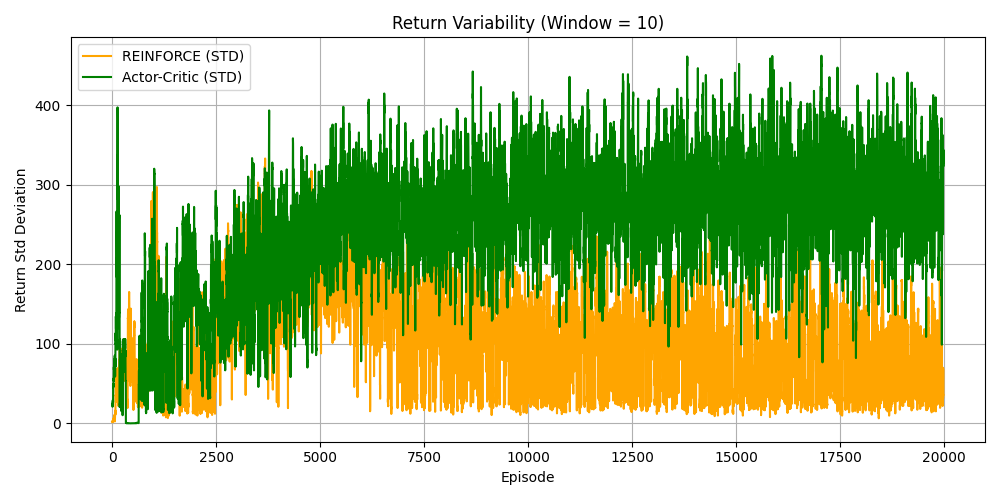
**Environment**

Observation space: continuous state vector Box([-inf -inf -inf -inf -inf -inf -inf -inf -inf -inf -inf], [inf inf inf inf inf inf inf inf inf inf inf], (11,), float64)

Action space: continuous joint torques Box([-1. -1. -1.], [1. 1. 1.], (3,), float32)

**Results** (20k steps)Immagine che contiene testo, Diagramma, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.



PPO

The previous methods have some problems, the updated policy can deviate too far from the old policy leading to poor performance or divergence.

This was solved by Trust Region Policy Optimization (TRPO) but it is too complex to implement.

PPO simplifies TRPO by introducing a clipped objective to control policy updates without complex constraints.

It uses:

The probability ratio of new vs old policy

The advantage estimate at time t

Clipped surrogate objective

This means that if the policy changes too much the update is clipped preventing over-optimization, it also ensures stable and conservative updates.

PPO can also train for multiple epochs using mini-batches improving sample efficiency

**Results** (1M steps)

No normalization (Source)

Immagine che contiene testo, Diagramma, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

With normalization (Source)

Immagine che contiene testo, Diagramma, linea, diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

With normalization (Target)

Immagine che contiene testo, Diagramma, diagramma, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Test on source env (no norm)

Immagine che contiene testo, schermata, diagramma, Diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Test on target env (no norm)

Immagine che contiene testo, schermata, diagramma, Diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Test on source env (with norm)

Immagine che contiene testo, schermata, diagramma, Diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

Test on target env (with norm)

Immagine che contiene testo, schermata, diagramma, Diagramma

Il contenuto generato dall'IA potrebbe non essere corretto.

SAC

It combines off-policy learning, entropy regularization, stochastic actor and twin q-learning.

SAC maximizes a modified objective that encourages exploration

Where is the entropy of the policy

And is the temperature parameter for controlling trade-off between reward and entropy

So, SAC tries to maximize both reward and entropy leading to stochastic policies that explore more broadly.

Components of SAC:

Soft Q-Function learns the expected return plus future entropy

The actor maximizes the soft value

SAC uses two Q-functions Q1 and Q2 and uses

Main steps of the algorithm:

Initialize:

-Replay buffer D

-Actor network

-Two Q-networks

-Target Q-networks

For each step

-Sample action

-Interact with environment, store in D

Sample mini-batch from D

Update Q-functions

-Compute target

-Minimize MSE loss

Update policy (actor)

Update temperature

Update target networks

SAC is particularly good when the action space is continuous, in sim-to-real scenarios and when high final performance and stability are required.

EXTENSIONS

Domain randomization is a useful tool to make a model robust to environment changes, unfortunately it follows a naïve approach and can end up overfitting the model on a sub-class of environments or at times even damage the model by training on unrealistic environments. We decided to implement two more advanced methods and compare how they perform.

The first, Reptile, is a simple and efficient first-order optimization-based meta-learning algorithm. Unlike more complex second-order methods such as Model-Agnostic Meta-Learning (MAML), Reptile avoids computing second derivatives while still enabling rapid adaptation to new tasks.

The core idea of Reptile is to train a model in such a way that a few steps of gradient descent on a new task (new environment) lead to good performance. This is achieved by explicitly optimizing for initial model parameters that are close to those that would result from fine-tuning on a single task, in other words it learns initial parameters that are easy to adapt.

Reptile operates by:

1. Sampling a task from a distribution over tasks
2. Performing several gradient steps using a standard optimizer on that task to obtain updated parameters
3. Moving the initial parameters slightly toward the updated one, this meta-update encourages the initial parameters to be more adaptable to a wide variety of tasks.

If θ is the initial parameter vector and θ′ is the adapted parameter vector after training on one task, Reptile performs θ←θ+ϵ(θ′−θ) where ϵ is a meta-learning rate.

In our implementation to compute θ′ we used PPO with 5000 steps, furthermore θ′ is the result of the average of multiple new parameters computed, this is we use an inner loop that iterates for a decided number of tasks, inside we train a model on a new task and in the end it averages the parameters of the trained models.

Results:

Better results were observed for a higher meta learning rate, higher number of meta updates, lower number of inner loops, and using a gaussian distribution instead of a uniform, this could be explained by the simplicity of the environment, training a fast learning model closer to the source environment and with higher learning rate can lead to good results when the target environment is not much different from the source environment. For harsher environment changes or more complex environments it could be useful to train with lower meta learning rate and different distributions to avoid overfitting.

The next method implemented is Adversarial Learning, it is a technique in machine learning where an agent, called the adversary, is trained with the goal of exposing weaknesses in another agent or system, often by creating difficult or challenging scenarios. In this context it is used to increase the robustness of a policy by training it against intentionally difficult environments variations. Rather than relying on random variations in environment parameters, adversarial learning involves training a second agent to actively search for perturbations that degrade the main agent’s performance. This dynamic interaction forms a two-player game, where the adversary tries to make the environment harder and the protagonist tries to succeed despite those challenges.

This method consists of two main stages:

1. Training the adversary, at each training step the adversary chooses a perturbation vector that adjusts the masses, a simulated episode is then run using a random policy in the perturbated environment, the adversary’s reward is the negative of the total return of the random agent (this encourages the adversary to learn perturbations that reduce the hopper’s performance), over time the adversary learns to identify mass configurations that are particularly challenging for control policies.
2. Training the protagonist, the adversary is used as part of a wrapper of the environment, during each episode the wrapper uses the adversary’s policy to apply targeted perturbations to the hopper’s body masses before training begins, the protagonist is then trained using PPO to perform well across these dynamically perturbated environments

The primary goal of adversarial learning in this context is to produce robust and generalizable policies that perform well not only in the nominal environment but also under a wide range of realistic physical variations

Results:

Comparison between meta learning, adversarial domain randomization and udr: