Bipedal Walker: A Challenging Benchmark for Continuous Control Algorithms

Pricop Tudor-Constantin

January 14, 2024

Abstract

Bipedal locomotion, a fundamental challenge in robotics, has gained significant attention in recent years due to its complexity and due to the emergence of advanced reinforcement learning (RL) techniques. In this study, we delve into the application of various RL algorithms to solve the BipedalWalker environment from OpenAI Gym, a popular benchmark for evaluating bipedal robot control strategies. We investigate the performance of Proximal Policy Optimization (PPO), Truncated Quantile Critics (TQC), Twin Delayed Deep Deterministic Policy Gradient (TD3), Soft Actor-Critic (SAC) and Augmented Random Search (ARS) algorithms in achieving stable and efficient bipedal walking. Our experimental results compare the algorithms across various metrics such as rewards or time complexity. Moreover, we analyze the learning dynamics of each algorithm, revealing their strengths and limitations. Our findings provide valuable insights into the application of RL techniques for solving complex control problems in robotics and suggest that there are promising approaches for training bipedal robots to walk effectively.

1 Introduction

Reinforcement Learning in the real world is still an ill-defined problem. The agent has to be greedy, but not too much. We want the agent the agent to be curious to exploit the environment whenever possible but not too curious to continue to work for us.

The BipedalWalker environment from Gym is a challenging problem to solve, having complex and sensitive model dynamics. For this adaptive robotic locomotion tasks with several degrees of freedom, the task becomes infeasible with classical control techniques. However, the robot's dynamics can be modeled and tested in numerous simulation trials using reinforcement learning to determine the optimal policy mapping the robot's state to the action required to walk forward.

The paper is structured as follows: we provide a detailed explanation of the problem to be optimized, its complexities and its relevance in real-world applications in the *Problem Description* section. Following, in the *Algorithms* section, we delve into the specifics of the RL algorithms used in this study, discussing the unique mechanisms and representations that have been tailored specifically for this locomotion environment. Next, the *Experiments* section describes the experimental design, including the benchmarks, the parameter settings for the algorithms and the performance metrics used to evaluate the quality of the solutions. In the *Results and Interpretation* section, we present and interpret the results of the experiments. We provide a comparative analysis of the RL algorithms' performance, assessing convergence rates, solution quality and computational efficiency. Finally, in the

Conclusion and Future Directions section, we summarize the key findings of the study, discuss their implications, suggest potential avenues for future research and provide concluding remarks. At the bottom, the *References* section includes the list of resources cited throughout the paper.

2 Problem Description

BipedalWalker-v3 is a challenging environment, the agent should run very fast, not trip itself off and use as little energy as possible. According to [1] the problem is described as follows, details mandatory for a better understanding of the approach taken.

2.1 Description

This is a simple 4-joint walker robot environment. There are two versions (**hardcore** parameter in environment constructor):

- Normal, with slightly uneven terrain.
- Hardcore, with ladders, stumps, pitfalls.

To solve the normal version, you need to get 300 points in 1600 time steps. To solve the hardcore version, you need 300 points in 2000 time steps. A simpler and more accessible variant of solving the environment would be to achieve an average total reward of over 300 over 100 consecutive episodes, as given in the 2nd version of the environment.

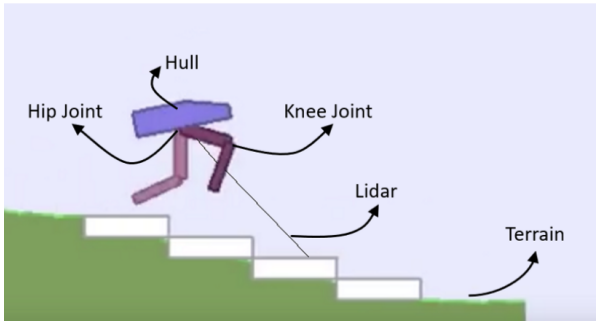

Figure 1: BipedalWalker agent - terminology

There are 4 primary strategies to walk. Usually, the agent tries all of them during the training process:

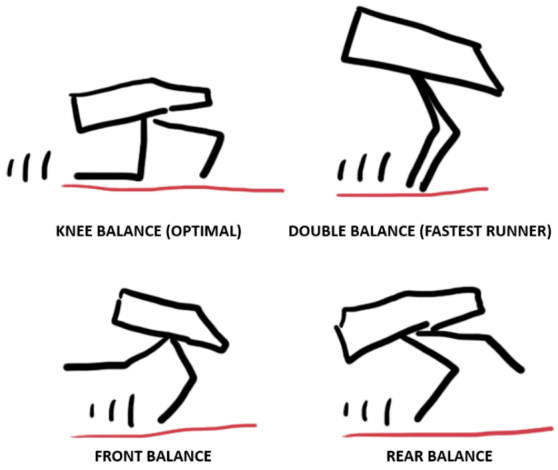

Figure 2: Walking strategies

2.2 Action Space

Actions are motor speed values for each of the 4 joints at both hips and knees.

Idx	Name	Min	Max
0	Hip 1 (Torque / Velocity)	-1	1
1	Knee 1 (Torque / Velocity)	-1	1
2	Hip 2 (Torque / Velocity)	-1	1
3	Knee 2 (Torque / Velocity)	-1	1

Action Space

2.3 Observation Space

State consists of hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements. There are no coordinates in the state vector.

Idx	Name	Min	Max
0	hull angle	-pi	pi
1	hull angular velocity	-5	5
2	vel x	-5	5
3	vel y	-5	5
4	hip joint 1 angle	-pi	pi
5	hip joint 1 speed	-5	5
6	knee joint 1 angle	-pi	pi
7	knee joint 1 speed	-5	5
8	leg 1 ground contact flag	0	5
9	hip joint 2 angle	-pi	pi
10	hip joint 2 speed	-5	5
11	knee joint 2 angle	-pi	pi
12	knee joint 2 speed	-5	5
13	leg 2 ground contact flag	0	5
14-23	10 lidar readings	-1	1

State Space

2.4 Rewards

Reward is given for moving forward, totaling 300+ points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points. A more optimal agent will get a better score.

2.5 Starting State

The walker starts standing at the left end of the terrain with the hull horizontal, and both legs in the same position with a slight knee angle.

2.6 Episode Termination

The episode will terminate if the hull gets in contact with the ground or if the walker exceeds the right end of the terrain length.

3 Algorithms

3.1 TQC - Truncated Quantile Critics

As presented in [5], the authors propose a new algorithm called Truncated Quantile Critics (TQC) that combines three techniques: distributional representation, truncation, and ensembling. TQC aims to improve off-policy learning in continuous control tasks by mitigating the overestimation bias that often plagues existing methods.

TQC represents the expected return as a distribution of atoms, rather than a single scalar value. By adjusting the number of atoms, it can control the precision of the return distribution approximation and the degree of overestimation.

The algorithm truncates the return distribution by dropping the atoms with the highest locations (i.e. the most optimistic estimates) and averaging the remaining ones. This reduces the overestimation bias, especially when the return variance is high. TQC can balance between underestimation and overestimation by varying the number of dropped atoms.

It ensembles multiple distributional critics by forming a mixture of their predictions and truncating it. This removes the outliers from the pool of all estimates and improves the robustness and accuracy of the Q-value estimation. TQC can also truncate each critic individually before mixing them, which may be useful in some scenarios.

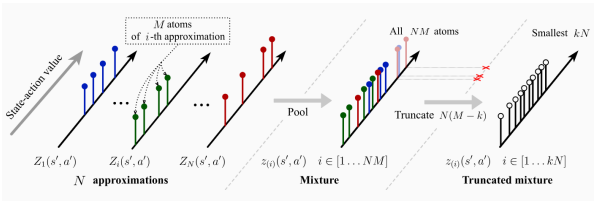

Figure 3: TQC explained

The method is compared with original implementations of state of the art algorithms: SAC, TrulyPPO and TD3 against a continuous control benchmark suite.

Figure shows the learning curves, and the results suggest that TQC performs consistently better than any of the competitors.

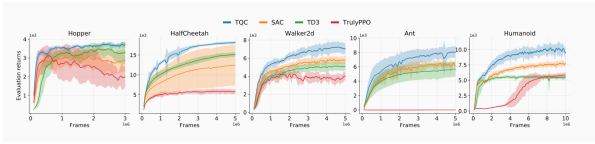

Figure 4: TQC results on several CC environments

3.2 TD3 - Twin Delayed Deep Deterministic Policy Gradient

TD3 is an extension of the Deep Deterministic Policy Gradient (DDPG) algorithm, which can be unstable and is heavily reliant on finding the correct hyper parameters for the current task. This problem is caused by the algorithm continuously over estimating the Q values of the critic (value) network. These estimation errors build up over time and can lead to the agent falling into a local optima or experience catastrophic forgetting. TD3 addresses this issue by focusing on reducing the overestimation bias seen in previous algorithms. In order to mitigate this issue, 3 key features have been introduced, which substantially improved the performance over baseline DDPG:

- **Clipped Double-Q Learning**: TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

- **"Delayed" Policy Updates**: TD3 updates the policy (and target networks) less frequently than the Q-function. The paper [6] recommends one policy update for every two Q-function updates.

- **Target Policy Smoothing**: TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Pseudocode:

Algorithm 1 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if j mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

Figure 5: TD3 pseudocode

3.3 PPO - Proximal Policy Optimization

The idea with PPO is that we want to improve the training stability of the policy by limiting the change made to the policy at each training epoch: we want to avoid having too large of a policy update, for 2 reasons:

- We know empirically that smaller policy updates during training are more likely to converge to an optimal solution.

- A too-big step in a policy update can result in falling "off the cliff" (getting a bad policy) and taking a long time or even having no possibility to recover.

Figure 6: Improve training stability by taking smaller updates

So with PPO, we update the policy conservatively. To do so, we need to measure how much the current policy changed compared to the former one using a ratio calculation between the current and former policy. And we clip this ratio in a range $[1-\epsilon, 1+\epsilon]$, meaning that we remove the incentive for the current policy to go too far from the old one (hence the proximal policy term).

PPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

Pseudocode:

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Figure 7: PPO pseudocode

3.4 SAC - Soft Actor-Critic

What makes an ideal deep RL algorithm for real-world systems? Real-world experimentation brings additional challenges, such as constant interruptions in the data stream, requirement for a low-latency inference and smooth exploration to avoid mechanical wear and tear on the robot, which set additional requirement for both the algorithm and also the implementation of the algorithm.

Regarding the algorithm, several properties are desirable:

- **Sample Efficiency**. Learning skills in the real world can take a substantial amount of time. Prototyping a new task takes several trials, and the total time required to learn a new skill quickly adds up. Thus good sample complexity is the first prerequisite for successful skill acquisition.

- **No Sensitive Hyperparameters**. In the real world, we want to avoid parameter tuning for the obvious reason. Maximum entropy RL provides a robust framework that minimizes the need for hyperparameter tuning.

- **Off-Policy Learning**. An algorithm is off-policy if we can reuse data collected for another task. In a typical scenario, we need to adjust parameters and shape the reward function when prototyping a new task, and use of

an off-policy algorithm allows reusing the already collected data.

Pseudocode:

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 8: SAC pseudocode

Soft actor-critic (SAC), described above, is an off-policy model-free deep RL algorithm that is well aligned with these requirements. In particular, it is shown that it is sample efficient enough to solve real-world robot tasks in only a handful of hours, robust to hyperparameters and works on a variety of simulated environments with a single set of hyperparameters.

3.5 ARS - Augmented Random Search

The authors from [7] assert that they have built an algorithm that is at least 15 times more efficient than the fastest competing model-free methods on MuJoCo locomotion benchmarks. They enhanced the existing Basic Random Search (BRS) with several adjustments, calling their discovery the Augmented Random Search.

The idea of Basic Random Search is to pick a parameterized policy π_θ, shock (or perturb) the parameters θ by applying $+\xi\delta$ and $-\xi\delta$ (where $\xi < 1$ is a constant noise and δ is a random number generated from a normal distribution).

Then apply the actions based on $\pi(\theta + \xi\delta)$ and $\pi(\theta - \xi\delta)$ then collect the rewards $r(\theta + \xi\delta)$ and $r(\theta - \xi\delta)$ resulting from those actions.

Now that we have the rewards of the perturbed θ, compute the average $\Delta = \frac{1}{N}\sum[r(\theta + \xi\delta) - r(\theta - \xi\delta)]\delta$ for all the δ and we update the parameters θ using Δ and a learning rate α: $\theta^{j+1} = \theta^j + \alpha.\Delta$.

The ARS is an improved version of BRS, containing 3 main axis of enhancements that increase its performance: **Dividing by the standard deviation**, **Normalizing the states** and **Using the top performing directions**.

As iterations go on, the difference between $r(\theta + \xi\delta)$ and $r(\theta - \xi\delta)$ can vary significantly, with the learning rate α fixed, the update $\theta^{j+1} = \theta^j + \alpha.\Delta$ might oscillate considerably. To avert this type of variations we divide $\alpha.\Delta$ by σ_r (Standard Deviation of the collected rewards).

The normalization of states ensures that policies put equal weight on the different components of the states. The normalization technique used in ARS consists of subtracting the current observed average of the state, from the state input and divide it by the standard deviation of the state: $\frac{(\text{state\_input} - \text{state\_observed\_average})}{\text{state\_std}}$

It would be useful to remember that our goal is to maximize the collected rewards. However we are computing the average reward in each iteration, meaning that in each iteration we compute $2N$ episodes each one following $\pi(\theta + \xi\delta)$ and $\pi(\theta - \xi\delta)$, then we average the collected rewards $r(\theta + \xi\delta)$ and $r(\theta - \xi\delta)$ for all the $2N$ episodes.

This presents some pitfalls because if some of the rewards are small compared to the others, they will push the average down.

One way remedy to this issue is to sort the rewards in the decreasing order based on the key $\max(r(\theta + \xi\delta), r(\theta - \xi\delta))$. Then use only the top b rewards (and their respective perturbation δ) in the computation of the average reward.

Pseudocode:

Algorithm 2 Augmented Random Search (ARS): four versions **V1**, **V1-t**, **V2** and **V2-t**

Hyperparameters: step-size α, number of directions sampled per iteration N, standard deviation of the exploration noise ν, number of top-performing directions to use b ($b < N$ is allowed only for **V1-t** and **V2-t**)

Initialize: $M_0 = \mathbf{0} \in \mathbb{R}^{p \times n}$, $\mu_0 = \mathbf{0} \in \mathbb{R}^n$, and $\Sigma_0 = \mathbf{I}_n \in \mathbb{R}^{n \times n}$, $j = 0$.

while ending condition not satisfied **do**

Sample $\delta_1, \delta_2, \ldots, \delta_N$ in $\mathbb{R}^{p \times n}$ with i.i.d. standard normal entries.

Collect $2N$ rollouts of horizon H and their corresponding rewards using the $2N$ policies

$$\textbf{V1:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)x \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)x \end{cases}$$

$$\textbf{V2:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)\,\mathrm{diag}\,(\Sigma_j)^{-1/2}(x - \mu_j) \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)\,\mathrm{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \end{cases}$$

for $k \in \{1, 2, \ldots, N\}$.

Sort the directions δ_k by $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$, denote by $\delta_{(k)}$ the k-th largest direction, and by $\pi_{j,(k),+}$ and $\pi_{j,(k),-}$ the corresponding policies.

Make the update step:

$$M_{j+1} = M_j + \frac{\alpha}{b\sigma_R} \sum_{k=1}^{b} \left[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-}) \right] \delta_{(k)},$$

where σ_R is the standard deviation of the $2b$ rewards used in the update step.

V2 : Set μ_{j+1}, Σ_{j+1} to be the mean and covariance of the $2NH(j+1)$ states encountered from the start of training.[2]

$j \leftarrow j + 1$

end while

Figure 9: ARS pseudocode

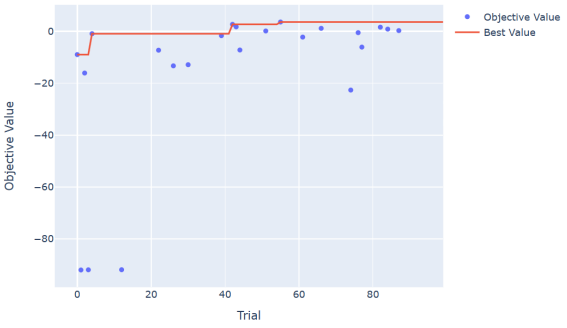

Figure 11: Optuna optimization trials

4 Experiments

4.1 Hyperparameters Tuning

When compared with Supervised Learning, Reinforcement Learning is far more sensitive to the choice of hyperparameters, poor choices leading to unstable convergence. Additionally, selecting the appropiate algorithm is also very important.

Optuna [8] is a library that helps to automate the search of finding a good set of training hyperparameters. It provides a variety of features, including basic optimization techniques, advanced pruning strategies, feature selection, and tracking experiment performance. It also offers built-in plots for visualizing optimization results and the ability to save optimization sessions for resuming or sharing across multiple processes. It uses state-of-the-art algorithms to efficiently search large spaces and prune unpromising trials for faster results.

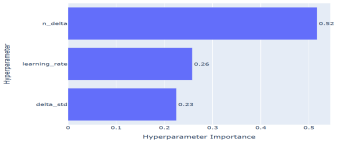

Figure 10: Optuna hyperparams importance

After running the optimization phase for the RL algorithms used, the results were captured in the following table:

Algorithm	Hyperparameters
ARS	learning_rate = 0.02 delta_std = 0.0075 n_delta = 64 n_top = 32 alive_bonus_offset = -0.1 normalize = norm_obs
PPO	n_steps = 2048 batch_size = 64 gae_lambda = 0.95 gamma = 0.999 n_epochs = 10 ent_coef = 0.0 learning_rate = 3e-4 clip_range = 0.18 normalize = norm_obs
SAC	buffer_size = 3e5 batch_size = 256 tau = 0.02 gamma = 0.98 train_freq = 64 ent_coef = auto learning_rate = 7.3e-4 gradient_steps = 64 learning_starts = 1e4 use_sde = True

Tuned hyperparameters

Algorithm	Hyperparameters
TD3	gamma = 0.98 buffer_size = 2e5 learning_starts = 1e4 noise_type = normal noise_std = 0.1 gradient_step = -1 training_freq = [1, episode] learning_rate = 1e-3
TQC	learning_rate = 7.3e-4 buffer_size = 3e5 batch_size = 256 ent_coef = auto gamma = 0.98 tau = 0.02 train_freq = 64 gradient_steps = 64 learning_starts = 1e4 use_sde = True

Tuned hyperparameters

4.2 Computational resources

Training RL agents require significant computational resources, depending on the complexity of the task (continuous control is a difficult task), the size of the state (24 continuous), the action space (4 continuous) and the specifics of the algorithm used. In general, the training involves a large number of episodes, each consisting of multiple steps of interaction with the environment. Furthermore, many RL algorithms are sample-inefficient, meaning they require a large number of episodes to converge to a good policy. Google Colab was used as one of the computational resources, however it had limited GPU capabilities and a time limit on the CPU, which was slow. Thus, another resource was used, which helped the training process: a personal device with a 3.00GHz CPU (11th Gen Intel(R) Core(TM) i7-1185G7) and 16 GB of RAM.

4.3 Training

We compare the performance of the above mentioned reinforcement learning algorithms. We use the stable-baselines3 library to implement these algorithms and we leverage the huggingface packages to upload the trained models to the Hugging Face Hub. We also use the Weights and Biases [12] library and Tensorboard metrics to track and visualize our experiments.

For each algorithm, the training was made on a vectorized environment, which stacks multiple independent environments, allowing the agent to train on multiple environments per step. However, due to the construction of some algorithms, some could train only on one environment at a time: ARS and TD3, while others were trained on 16 stacked environments: PPO, SAC and TQC. Furthermore, it was experimented with both normalizing the observations and without normalization and the best results were pushed to the Hub (for the normalized agents, the normalizer with the parameters used for training must also be used at evaluation in order to get the same results).

The training pipeline also offers the possibility to load the models from the Hub and resume their training, as one can't know from the beginning how many epochs are needed for the agent to converge to a good policy. Finally, the agent's policy is evaluated and the results, mean and standard deviation of the episodic rewards, along with tensorboard statistics are pushed to the Hub.

While Stable Baselines has an ARS training algorithm, it is located inside the Experimental Library [11], and trying several combinations of parameters did not lead to any conclusive results (the agent could not learn a stable policy). Thus, a personal implementation of the ARS has been made, including the normalization of the environment, the monitoring of the agent and the metrics, and the pushing to the Hub method.

5 Results and Interpretation

The performance of the Bipedal Walker agent is measured by several metrics and showcases competitive results.

Firstly, as described in the solving requirement of the 3rd version of the environment, the metric encompasses both the total reward and the episode

length. The total reward is the cumulative sum of the rewards obtained by the agent at each time step, which depend on the distance traveled, the energy spent and the stability of the walker. The episode length is the number of time steps that the agent can sustain the walking behavior before falling or reaching the end of the terrain. The goal of the agent is to maximize the total reward and the episode length, while avoiding excessive energy consumption and instability.

Another metric used by Huggingface Deep RL Leaderboard [13], which is more competitive, is the ability to obtain not only a score of over 300, but to obtain as much as possible, surpassing other users' scores. The score is computed by subtracting the standard deviation of the evaluation rewards from their mean. In order to check the results, please search by user id **MadFritz**.

Of course, the human evaluation is also an important metric: each of the models pushed to Huggingface contain a video demonstration of their walking capability, allowing us to analyze the behaviours, the strategies learnt and what are the most difficult states inside the environment.

Lastly, perhaps the most difficult metric would be the speed of learning, meaning how many epochs are required until the agent converged to a policy with satisfying cumulative rewards, over a mean of 300. Such a leaderboard is published by OpenAI [14] and the fastest implementations seem to contain additional features over the state-of-the-art vanilla RL algorihms presented above.

5.1 Bipedal Walker Simple

Regarding the first metric, it seems that all methods were able to reach a score of over 300, while not making more than 1600 steps per episode, meaning they successfully solved the environment, Tensorboard statistics for each of the methods can be checked in the *Metrics* tab of the Model repository inside Hugginface Hub. However, for some models not all Tensorboard sessions have been succesfully pushed, perhaps because of the train/resume cycle. Check here all the repositories: TQC[1], PPO[2], ARS[3], TD3[4], SAC[5].

Rank	Algorithm	Score	Mean	Std
3	TQC	328.78	329.11	0.33
6	PPO	319.96	320.69	0.73
8	SAC	318.07	318.72	0.65
10	ARS	312.40	329.08	16.68
14	TD3	305.59	306.79	1.20

BipedalWalker scores on Huggingface Leaderboard

Next, inside the *Model card* tab of the Model repository inside Huggingface Hub, one can analyze the agent playing in the environment. While the ARS agent seems to learn the optimal strategy: the Knee Balance, the others try to hurry, learning to run as fast as they can: the Double Balance strategy.

Algorithm	Iterations until reaching 300
ARS	1K
TD3	500K
TQC	700K
SAC	1M
PPO	3M

Approximative number of iterations before reaching 300 mean reward

As expected, ARS excels in finding a good policy in a small number of iterations.

5.2 Bipedal Walker Hardcore

The hardcore version of this enironment is indeed hardcore, as none of the agents were able to get to a score of 300. In fact, only one of them managed to understand a bit how to take over the obstacles, using a limited resources budget. Check their

[1] https://huggingface.co/MadFritz/tqc-BipedalWalker-v3
[2] https://huggingface.co/MadFritz/ppo-BipedalWalker-v3
[3] https://huggingface.co/MadFritz/ars-BipedalWalker-v3
[4] https://huggingface.co/MadFritz/td3-BipedalWalker-v3
[5] https://huggingface.co/MadFritz/sac-BipedalWalker-v3

repositories: ARS[6], TD3[7], PPO[8], SAC[9], TQC[10].

Unfortunately, there is no Huggingface Leaderboard for the hardcore version, but the personal best agent, TQC, was compared with the best performing agent from stable-baselines3 account[11] and it managed to obtain a better score.

Algorithm	Score	Mean	Std	sb3
TQC	158.68	244.20	85.52	86.67
SAC	-44.05	13.87	57.92	-96.11
PPO	-75.48	8.23	83.71	-72.07
TD3	-113.91	-86.32	27.59	-113.93
ARS	-133.32	-97.25	36.07	-

BipedalWalkerHardcore scores

As seen, the results are not at all close to having the environment solved. Maybe with a big enough budget (number of iterations and computational resources) the agents will be able to converge, however for fast results there must be additional powerful techniques included in the algorithm, mechansims specifically tailored for this type of environment.

Visually, ARS, TD3 and PPO are trying to double balance the agent (until it falls or gets stuck), while SAC and TQC found that the best balancing method is on the rear leg.

Additional metrics can be analyzed[16]: statistics with parameters from each run from the train/resume cycle, videos with the progress along the runs and also Memory and Network consumption data, useful to assess whether more powerful resources are needed and if more environments can be parallelized for training.

6 Conclusion and Future Directions

When choosing a RL method to implement in a system, there is no all-encompassing algorithm that universally produces the best results. It is necessary to not only consider the structure of the problem, but also experiment with various algorithms and various hyperparameters for those algorithms.

In context of the Bipedal Walker, the realm of possible algorithms to choose from is narrowed due to the large continuous state space and action space, producing the best results with a model-free method. While with some continuous problems it is possible to discretize the state space and action space, doing so is computationally intractable for theproblem at hand since producing a sufficiently granular state space and action space would result in memory overflow as the discretized space scales exponentially with bin size.

While the state-of-the-art algorithms presented in the paper worked fine with the normal version of the environment, it seems that the hardcore version still poses a challenging problem. Future work could involve either trying better versions of the vanilla RL algorithms, like FORK[15], which was proven to be able to solve the environment in around 3500 episodes, either leveraging the fact that the agent has already learnt to walk on flat ground (normal BipedalWalker) and it should only explore the ways in which it can overcome the new obstacles.

References

[1] Problem description: Gym website `https://gymnasium.farama.org/environments/box2d/bipedal_walker/`

[2] Problem description and solution `https://arxiv.org/pdf/2112.07031.pdf`

[3] Huggingface Deep RL Course `https://huggingface.co/learn/deep-rl-course/unit0/introduction`

[4] Soft Actor Critic `https://bair.berkeley.edu/blog/2018/12/14/sac/`

[6] https://huggingface.co/MadFritz/ars-BipedalWalkerHardcore-v3
[7] https://huggingface.co/MadFritz/td3-BipedalWalkerHardcore-v3
[8] https://huggingface.co/MadFritz/ppo-BipedalWalkerHardcore-v3
[9] https://huggingface.co/MadFritz/sac-BipedalWalkerHardcore-v3
[10] https://huggingface.co/MadFritz/tqc-BipedalWalkerHardcore-v3
[11] https://huggingface.co/sb3

[5] Truncated Quantile Critics paper https://arxiv.org/abs/2005.04269

[6] Twin Delayed DDPG paper https://arxiv.org/pdf/1802.09477.pdf

[7] Augmented Random Search paper https://arxiv.org/pdf/1803.07055.pdf

[8] Hyperparameter tuning with Optuna https://optuna.org/

[9] Stable Baselines3 https://stable-baselines3.readthedocs.io/en/master/

[10] RL Baselines3 Zoo https://github.com/DLR-RM/rl-baselines3-zoo/tree/master

[11] Stable Baselines3 Contrib Package https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

[12] Weights and Biases https://wandb.ai/site

[13] Huggingface Deep RL Leaderboard for BipedalWalker environment https://huggingface.co/spaces/huggingface-projects/Deep-Reinforcement-Learning-Leaderboard

[14] Gym Leaderboard for BipedalWalker-v2 environment https://github.com/openai/gym/wiki/Leaderboard#bipedalwalker-v2

[15] Forward-Looking Actor for Model-free RL https://arxiv.org/pdf/2010.01652.pdf

[16] Metrics monitored with Weights and Biases

- https://wandb.ai/pricoptudor/ppo_BipedalWalker-v3
- https://wandb.ai/pricoptudor/tqc_BipedalWalker-v3
- https://wandb.ai/pricoptudor/td3_BipedalWalker-v3
- https://wandb.ai/pricoptudor/sac_BipedalWalker-v3
- https://wandb.ai/pricoptudor/ars_BipedalWalker-v3
- https://wandb.ai/pricoptudor/tqc_BipedalWalkerHardcore-v3
- https://wandb.ai/pricoptudor/td3_BipedalWalkerHardcore-v3
- https://wandb.ai/pricoptudor/ppo_BipedalWalkerHardcore-v3
- https://wandb.ai/pricoptudor/sac_BipedalWalkerHardcore-v3