# Solving the BipedalWalker-v3 gym environment

Apostu Daniel MISS12
Samson Ioan-Paul MISS12

## 1. Abstract

In this article we will show the methods we used for solving the Bipedal Walker gym environment, as well as make comparisons with what others have done. This article explores the application of the Twin Delayed DDPG (TD3) algorithm. Reinforcement learning forms the core of our exploration, a paradigm where an agent learns to make decisions by interacting with an environment. The interactions are based on the output of the Q function, denoted Q(s,a), where “s” is the state and “a” is the output action. This Q function essentially guides decision-making, influencing how the agent navigates its environment. In the realm of deep reinforcement learning, a noteworthy advancement is Deep Q-learning. This technique incorporates deep neural networks to approximate the Q-function, enabling better decision-making processes. While DQN is effective in discrete action environments, the Deep Deterministic Policy Gradient (DDPG) algorithm introduces a methodology that extends the applicability of deep reinforcement learning to problems where actions are continuous and potentially high-dimensional. Moreover, the Twin-Delayed DDPG (TD3) algorithm is an extension of DDPG, using two critic networks to estimate the value of the Q function, then taking the minimum value between a pair of critics to limit over-estimation.

## 2. Introduction

### 2.1 The problem

The Bipedal Walker gym environment presents a challenge in the field of reinforcement learning, focusing on the normal version of the scenario. In this environment, the objective is to train an intelligent agent to skillfully control a 4-joint walker robot through terrains featuring slightly uneven ground. The agent must accumulate an average of 300 points within a time frame of 1600 steps for 100 episodes to successfully solve the

problem. Actions are represented as values for the 4 joints, constrained to the [-1, 1] range. The observation space includes hull angle speed, angular velocity, horizontal and vertical speeds, joint positions, angular speed, legs' contact with the ground, and lidar rangefinder measurements, with no explicit coordinates in the state vector. Agents receive positive rewards for forward movement, accumulating up to 300 points, but falling incurs a penalty of -100 points. Applying motor torque also costs a small amount of points, incentivizing agents to optimize their strategies. Each episode begins with the walker standing at the left end of the terrain with a horizontal hull and both legs in the same position. Episodes terminate if the hull contacts the ground or if the walker exceeds the right end of the terrain length.

## 2.2 Our solution

Our solution is built on the TD3 algorithm which, as mentioned, is an extension of the DDPG algorithm.

DDPG employs actor and critic networks to facilitate learning in continuous action spaces. The actor network, denoted by $\mu(s|\theta)$, represents the policy and is responsible for determining the optimal action to take given a certain state. On the other hand, the critic network, $Q(s,a|\phi)$, estimates the action-value function and evaluates the chosen action's effectiveness in a given state. The actor network is trained to maximize the expected action-value, while the critic network is trained to minimize the mean squared Bellman error between the predicted Q-values and the computed targets. These networks work collaboratively, with the actor guiding the decision-making process and the critic providing feedback on the chosen actions' expected values.

Similar to DDPG, Twin Delayed DDPG (TD3) operates with actor and critic networks to navigate continuous action spaces. TD3 builds upon the foundations of DDPG with a few key modifications aimed at enhancing overall performance.
The first notable change in TD3 is related to the action-value function. In order to mitigate overestimation biases, TD3 employs two action-value function approximators, i.e. two critic networks. These critics optimize for the same objective, and when evaluating targets, TD3 takes the element-wise minimum between the two to prevent overly optimistic estimates.

The second modification involves delaying the updating of policy and target parameters, including $\theta$, $\phi_{target1}$,$\phi_{target2}$, and $\theta_{target}$t. This delay allows the Q-function parameters to stabilize before being treated as constants during the improvement of policy parameters and updating of target parameters. This deliberate delay contributes to the "Delayed" aspect in the name, Twin Delayed DDPG.

The final significant change in TD3 is the introduction of noise to the target actions before computing the targets. This addition aims to prevent overfitting to narrow peaks in the value estimate, ensuring that similar state-action pairs yield comparable action-value estimates. This modification helps enhance the robustness of the learning process, promoting more generalized policies in the face of environmental uncertainties.

# 3. State of the art

## 3.1 What other people did

In exploring the landscape of solving the Bipedal Walker gym environment and similar reinforcement learning challenges, we draw inspiration from various seminal works and noteworthy contributions. It's crucial to acknowledge the collective efforts of the research community in advancing methodologies for training intelligent agents.

There is a whole github topic dedicated to reinforcement learning, and specifically to solving this environment, and a leaderboard.

People have attempted solving the problem using different RL algorithms, such as DDPG, TD3, PPO, SAC etc. and evolutionary algorithms such as ARS. For algorithms such as TD3 or ARS people have been able to solve the environment (TD3~1000 episodes, ARS~100 episodes), and some that don't solve it quickly (DDPG~2000 episodes).

Furthermore, we studied papers for this assignment, some helpful ones being Addressing Function Approximation Error in Actor-Critic Methods and Taming the Estimation Bias in Deep Reinforcement Learning.

# 4. Our implementation

For the TD3 algorithm we created the Double Deep Q-network, consisting of one actor network, an actor-target network, two critic networks and two critic-target networks.

```python
class Actor(nn.Module):
    l1: nn.Linear
    l2: nn.Linear
    l3: nn.Linear
    max_action: float

    def __init__(self, state_size, action_size, max_action):
        super(Actor, self).__init__()

        self.l1 = nn.Linear(state_size, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, action_size)

        self.max_action = max_action

    def forward(self, state):
        x = F.relu(self.l1(state))
        x = F.relu(self.l2(x))
        x = torch.tanh(self.l3(x)) * self.max_action
        return x
```

```python
class Critic(nn.Module):
    l1: nn.Linear
    l2: nn.Linear
    l3: nn.Linear

    def __init__(self, state_size, action_size):
        super(Critic, self).__init__()

        self.l1 = nn.Linear(state_size + action_size, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, 1)

    def forward(self, state, action):
        state_action = torch.cat([state, action], 1)

        Q = F.relu(self.l1(state_action))
        Q = F.relu(self.l2(Q))
        Q = self.l3(Q)
        return Q
```

The separation of critic networks into critics and target critics, as well as the actor network into actor and target actor, is a key component in improving the stability and convergence of algorithms like DDPG (Deep Deterministic Policy Gradients) and its variants like TD3 (Twin Delayed DDPG). This separation is part of the technique known as target networks.

The split between online critics and target critics in the Twin Delayed DDPG (TD3) algorithm serves two crucial purposes. First, it introduces stability by utilizing target critics with delayed updates. This prevents rapid changes in the Q-value estimates during training, offering a more reliable and robust learning process. Second, the twin critics architecture, where the minimum Q-value between the two is taken, mitigates overestimation biases and variance, contributing to more accurate and conservative action-value estimates. This separation ensures that the critic networks track a slowly evolving target, reducing the risk of oscillations and enhancing the overall learning stability.

The separation of the actor network into an online actor and a target actor is essential for policy optimization in TD3. The target actor provides a stable reference for

guiding the policy during training, preventing abrupt changes and promoting smoother convergence. By delaying the updates to the target actor parameters, the algorithm ensures a consistent and slowly changing target for the actor-critic optimization. This separation facilitates a more reliable learning process, as the policy adjustments are made gradually, preventing premature and potentially destabilizing updates. The use of target actors, along with the coordinated soft updates, contributes to the stability and effectiveness of the actor-critic learning process in continuous action space environments.

```python
def update(self, memory, current_timestep, batch_size=100,
           gamma=0.99, tau=0.995, noise=0.2, noise_clip=0.5, policy_delay=2):
    for i in range(current_timestep):
        state, action, reward, next_state, done = memory.sample(batch_size)
        state = torch.FloatTensor(state).to(device)
        action = torch.FloatTensor(action).to(device)
        reward = torch.FloatTensor(reward).reshape((batch_size, 1)).to(device)
        next_state = torch.FloatTensor(next_state).to(device)
        done = torch.FloatTensor(done).reshape((batch_size, 1)).to(device)

        noise = torch.FloatTensor(action).data.normal_(0, noise).to(device)
        noise = noise.clamp(-noise_clip, noise_clip)
        next_action = (self.actor_target(next_state) + noise)
        next_action = next_action.clamp(-self.max_action, self.max_action)

        target_Q1 = self.critic_1_target(next_state, next_action)
        target_Q2 = self.critic_2_target(next_state, next_action)
        target_Q = torch.min(target_Q1, target_Q2)
        target_Q = reward + ((1 - done) * gamma * target_Q).detach()

        Q1 = self.critic_1(state, action)
        loss_Q1 = F.mse_loss(Q1, target_Q)
        self.critic_1_optimizer.zero_grad()
        loss_Q1.backward()
        self.critic_1_optimizer.step()

        Q2 = self.critic_2(state, action)
        loss_Q2 = F.mse_loss(Q2, target_Q)
        self.critic_2_optimizer.zero_grad()
        loss_Q2.backward()
        self.critic_2_optimizer.step()
```

```python
    if i % policy_delay == 0:
        actor_loss = -self.critic_1(state, self.actor(state)).mean()

        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        for actor_param, actor_target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
            new_actor_param_data = (tau * actor_target_param.data) + ((1 - tau) * actor_param.data)
            actor_target_param.data.copy_(new_actor_param_data)

        for critic_1_param, critic_1_target_param in zip(self.critic_1.parameters(),
                                                        self.critic_1_target.parameters()):
            new_critic_1_param_data = (tau * critic_1_target_param.data) + ((1 - tau) * critic_1_param.data)
            critic_1_target_param.data.copy_(new_critic_1_param_data)

        for critic_2_param, critic_2_target_param in zip(self.critic_2.parameters(),
                                                        self.critic_2_target.parameters()):
            new_critic_2_param_data = (tau * critic_2_target_param.data) + ((1 - tau) * critic_2_param.data)
            critic_2_target_param.data.copy_(new_critic_2_param_data)
```

The TD3 algorithm works as follows:

Firstly, random batches of experiences (state, action, reward, next_state, done) are sampled from the replay memory. Next, noise is added to the sampled action to encourage exploration. The noise is clipped to ensure it stays within a specified range.

The target actor network is used to predict the next action in the next state, incorporating the added noise. This action is clipped to ensure it stays within the valid action space.

Target Q-values are computed using the target critics for the next state and the predicted next action. The minimum Q-value between the twin critics is taken as the target Q-value. This helps mitigate overestimation biases. The online critic networks (self.critic_1 and self.critic_2) are updated using mean squared error loss between the predicted Q-values and the computed target Q-values. Backpropagation and optimization are performed separately for each critic.

After the policy_delay, target networks (actor and critics) are updated using a soft update (or polyak update) strategy. This involves a weighted combination of the target and online network parameters, controlled by the parameter $\tau$.

```
for episode in range(START_EPISODE, MAX_EPISODES + 1):
    state = env.reset()
    state = state[0]

    for timestep in range(MAX_STEPS_PER_EPISODE):
        action = agent.select_action(state)
        action = action + np.random.normal(0, EXPLORATION_NOISE, size=env.action_space.shape[0])
        action = action.clip(action_low, action_high)

        next_state, reward, done, _, _ = env.step(action)
        memory.save_experience((state, action, reward, next_state, float(done)))
        state = next_state

        average_reward += reward
        episode_reward += reward

        # if episode is done then update agent:
        if done or timestep == (MAX_STEPS_PER_EPISODE - 1):
            agent.update(memory, timestep, batch_size=BATCH_SIZE, gamma=GAMMA, tau=TAU,
                         noise=NOISE, noise_clip=NOISE_CLIP, policy_delay=POLICY_DELAY)
            break
```

During the training process, the provided code represents the central loop where the reinforcement learning agent, implementing the Twin Delayed DDPG

(TD3) algorithm, undergoes iterative interactions with the environment across episodes. At the start of each episode, the environment is reset, and the initial state is extracted. Subsequently, the agent selects actions based on the current state, incorporating exploration noise to encourage diversity in its actions. The chosen action is then applied to the environment, resulting in the next state, a reward, and an indication of whether the episode is concluded. After the episode is finished, the TD3 agent is updated.

Each 50 episodes after episode 300 the network parameters are saved. Also, they are saved if the algorithm has solved the environment.

## 4.1 Benchmark Performance

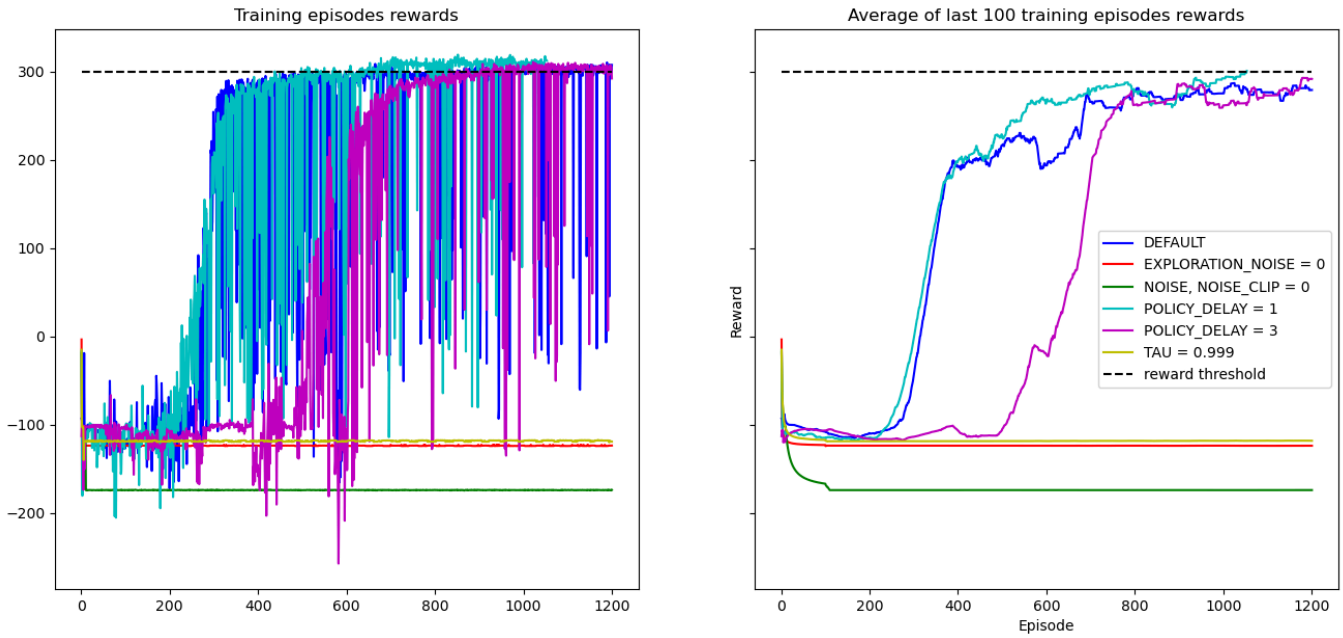
The default hyperparameters for the algorithm are the ones described in the TD3 paper. Both the actor and the critics consist of a two layer feedforward neural network of ***400 and 300 hidden nodes***, with ***ReLU units*** between each layer for both, ***and a final tanh unit after the output of the actor***. The critic receives both the state and action as input in the first layer, but both use ***Adam*** as the optimizer and a ***learning rate of 10^-3***. After each step, the networks are trained with a ***minibatch of 100*** transitions sampled uniformly from a replay buffer containing the history of the agent.
Target policy smoothing is implemented by adding a ***noise of N(0, 0.2)*** to the actions of the target actor network, ***clipped to (-0.5, 0.5)***.
Delayed policy updates consider updating the actor and target critic network only every ***d*** iterations, with ***d = 2***. Both the target networks are updated using a soft update with ***TAU = 0.005.***
To reduce the impact of the initial parameters of the policy the ***first 1000 timesteps*** are used only for exploration, using an off-policy exploration strategy adding ***Gaussian noise N(0, 0.1).***

With these parameters we managed by **episode 850 to reach a 301.924 average score in testing, with all episodes above 300 score and between 750 and 800 timesteps per episode**, putting us in the **middle of the leaderboard**.

| User | Version | Episodes before solve | Write-up | Video |
|---|---|---|---|---|
| Benjamin & Thor | 3.0 | 57 (TRPO with OU action noise) | writeup | |
| timurgepard | 3.0 | 70 (Symphony 🎹 ver 2.0) | writeup | video |
| timurgepard | 3.0 | 100 (Monte-Carlo 🌊 & Temporal Difference 🔥) | writeup | |
| Lauren | 2.0 | 110 | writeup | Video |
| Mathias Åsberg 😍 | 2.0 | 164 | writeup | Video |
| liu | 2.0 | 200 (AverageEpRet:338) | writeup | |
| Nandino Cakar | 3.0 | 474 | writeup | |
| Yoggi Voltbro | 3.0 | 696 | write-up | video |
| Nikhil Barhate | 2.0 | 800 | writeup | gif |
| Nick Kaparinos | 3.0 | 800 | Write-up | gif |
| Vinit & Abhimanyu | 2.0 | 910 | writeup | Video |
| shnippi | 3.0 | 925 | writeup | |
| M | 2.0 | 960 | writeup | Video |
| mayurmadnani | 2.0 | 1000 | Write-up | Youtube |
| Rafael1s | 2.0 | 1795 | Write-up | Youtube |
| chitianqilin | 2.0 | 47956 | writeup | Youtube |
| ZhiqingXiao | 3.0 | 0 (use close-form preset policy) | writeup | |
| koltafrickenfer | 2.0 | N/A | writeup | youtube |
| alirezamika | 2.0 | N/A | writeup | |
| 404akhan | 2.0 | N/A | writeup | |
| Udacity DRLND Team | 2.0 | N/A | writeup | gif |

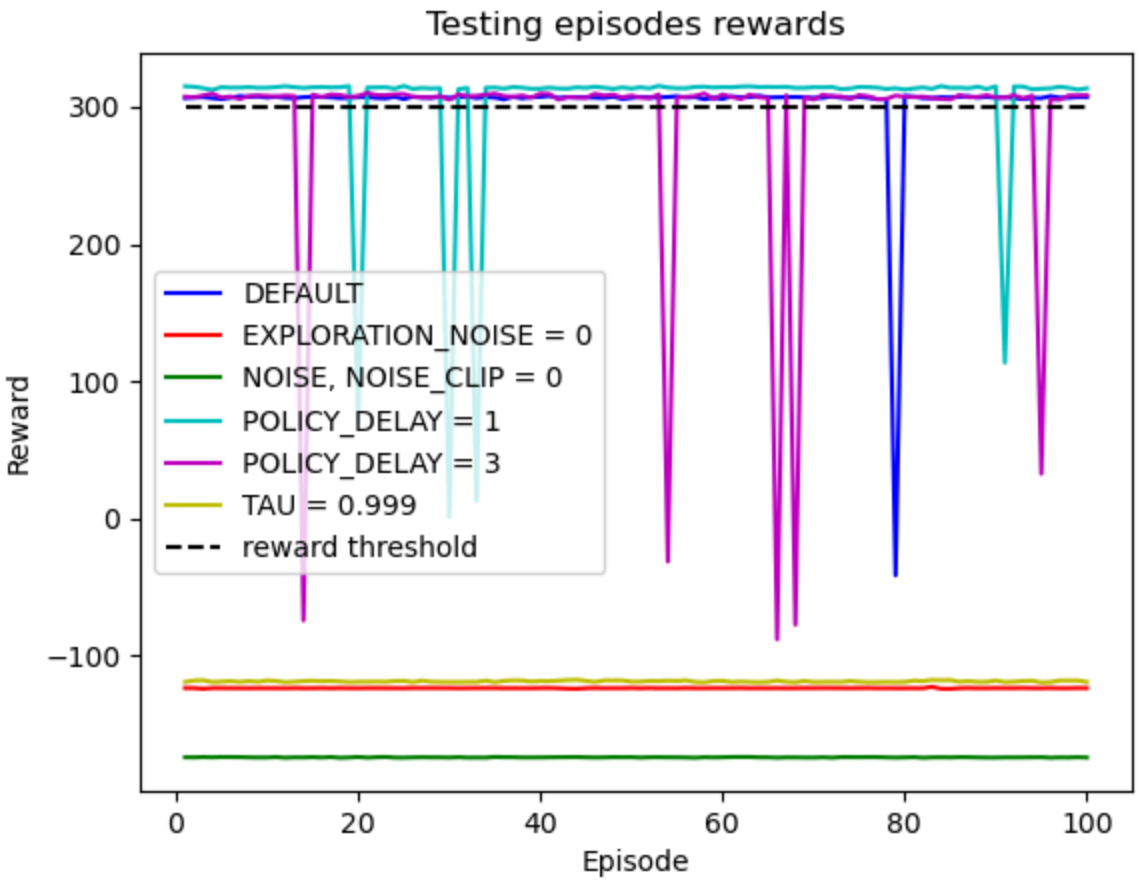
## 4.2 Our contributions / Ablation study

In our work we tried to deviate from the hyperparameters described in the [TD3 paper] to see how they impact learning and stability, both in training and testing.




1. Plots showing the reward per episode and the average reward of the last 100 episodes respectively of the last 100 episodes.

Here we can see that for the BipedalWalker-v3 environment, removing the policy smoothing noise, or removing the exploration noise, or softening the update even more, by making TAU smaller, renders the algorithm completely useless, as neither of these configurations are able to learn enough to at least get above 0 reward, they just plateau at a negative reward value.
On the other hand, modifying the value for the delayed updates, from d = 2 to d = 1 and d = 3 respectively does not wreck the learning curve, in the case of the d = 1, on some portions learning actually occurs faster, and we also have slightly higher values during testing.
But the reason as to why the authors choose d = 2 was to reduce the variance of the learned policy, and that can be seen on the testing plot, where the default policy experienced just one major dip, but the ones with POLICY_DELAY = 1 AND POLICY_DELAY = 2 have experienced several.




2. Plot showing the reward values during testing for 6 hyperparameter sets.

# 5. Conclusions

During the course of this project we managed to improve our knowledge over Reinforcement Learning algorithms by exploring different options. The field and competitions are still active, as the first place on the leaderboard was taken just 5 days ago, by an approach that managed to solve the environment in just 57 episodes. Even though we did not achieve a very high result, we are happy that we managed to beat our score from last year, when we took the ARS approach for the BipedalWalker-v2 environment, and managed to write a more detailed analysis, by testing different parameters. We conclude that the default hyperparameters are optimal and to improve our results in the future, we should consider other algorithms.

# 6. Video