A Beginner Tutorial for Deep Deterministic Policy Gradient (DDPG) Method in Reinforcement Learning

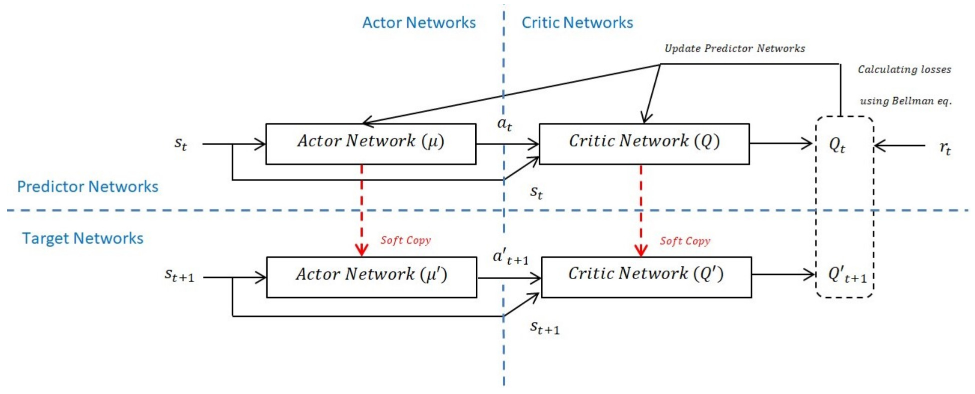
Alireza Khaleghi

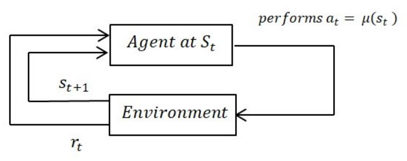
Reinforcement Learning for Bioengineers

Department of Bioengineering

Imperial College London

Autumn 2024





1 Introduction

This tutorial serves as the second course for the Reinforcement Learning for Bioengineers module, focussing on the Deep Deterministic Policy Gradient (DDPG) method. DDPG is an off-policy Actor-Critic algorithm in Deep Reinforcement Learning designed for environments with continuous action spaces. This feature makes DDPG particularly advantageous for real-world applications, such as in robots, where the robot should act and be controlled continuously to achieve high precision in manipulation.

The DDPG method employs two key components: actor and critic networks. The actor network takes the environment's states as input and generates continuous actions for the agent. These actions, combined with the states, are fed into the critic network, which estimates the Q-values. The Q values of the critic are compared with the Bellman target Q values to compute an error, which is used to optimise both networks.

DDPG incorporates target networks for both the actor and the critic to address instability in Q-value updates. These target networks update more gradually than their predictor counterparts, smoothing the learning process and enhancing stability. In conclusion, the DDPG algorithm needs four networks to be implemented: the predictor actor, the predictor critic, the target actor, and the target critic. The first two are being updated based on real-time errors for training purposes, and the last two are being soft-copied of the first two networks for stability purposes.

1.1 DDPG in Bioengineering

As highlighted earlier, the DDPG method is widely used in robotics and control applications where precise command execution is essential. In bioengineering, one notable application of DDPG is in the development of personalised prosthetics. Since each individual exhibits a unique gait pattern, it is challenging for prosthetics to adapt to every patient. However, by utilising model-free reinforcement learning methods like DDPG, prosthetics can learn and adapt to an individual's gait pattern, thereby improving customisation and performance (Khamies et al., 2024).

Another notable application of DDPG in bioengineering is in bipedal robots. The human gait cycle is highly complex, making it difficult for robots to replicate. As a result, achieving stable walking for robots is a challenging task. Although human motor learning and adaptation involve sophisticated processes, robots can leverage continuous reinforcement learning algorithms such as DDPG to stabilise their gait and achieve more human-like walking behaviour (Kumar et al., 2024).

2 Theoretical Explanation

2.1 The Overall Review of Method and Notations

This section is primarily based on the publication Continuous Control with Deep Reinforcement Learning by Lillicrap, Hunt, Pritzel, Heess, Erez, Tassa, Silver and Wierstra (2016).

First, the Bellman equation for the Temporal Difference (TD) approach should be briefly explained. This equation is used to estimate Q values, which serve as a baseline to calculate errors by comparing them to the Q values generated by the critic network. The Bellman equation is expressed as follows:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a_t) \tag{1}$$

Here, $Q(s_t, a_t)$ represents the Q-value of taking action a_t in the state s_t at time t; $r(s_t, a_t)$ is the immediate reward obtained from acting a_t in the state s_t; γ is the discount factor that reduces the significance of long-term rewards; and $Q(s_{t+1}, a_t)$ is the Q-value of the next state s_{t+1}. The term $Q(s_{t+1}, a_t)$ should be calculated using the target network since normally we do not have access to the data for the next state.

Before delving into the primary algorithm, it is better to define the notation first. The actor networks are shown with μ, and the critic networks are indicated with Q notation. The weights of the actors and the critic networks are shown by θ^μ and θ^Q, respectively. The target networks are assigned with prime symbols, Q' and μ' respectively.

The input of the actor network is the states at time t, hence we can write $\mu = \mu(s_t|\theta^\mu)$ and the output of the actor network is the continuous action at time t, a_t.

$$a_t = \mu(s_t|\theta^\mu) \tag{2}$$

The action a_t generated by the actor network combined with the state at time t (s_t)is directly fed into the critic network. Therefore, we can write $Q = Q(s_t, a_t|\theta^Q) = Q(s_t, \mu(s_t)|\theta^Q)$. Figure 1 illustrates the schematic of the overall DDPG method.

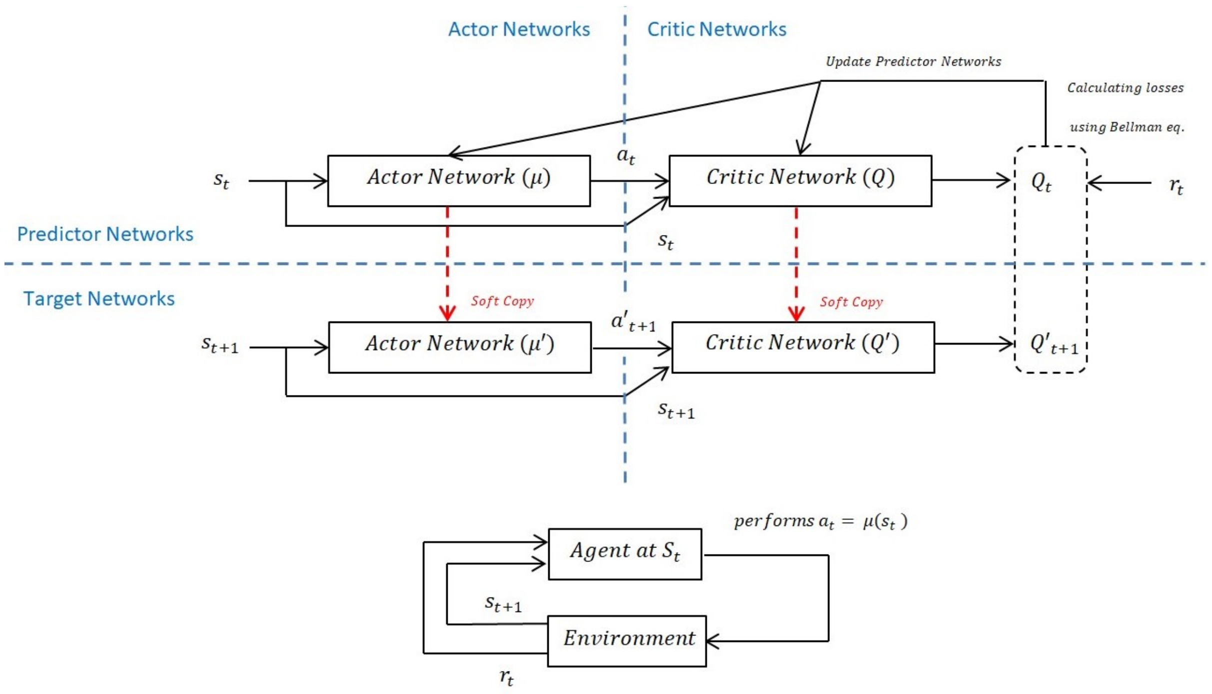


Figure 1: The Schematic of the DDPG Method

As shown in Figure 1, the network of predictor actors determines the action a_t. The agent performs this action (a_t) at time t in the environment, resulting in a reward r_t and a transition to the next state s_{t+1}. This interaction is recorded as an experience tuple (s_t, a_t, r_t, s_{t+1}) and stored in a replay buffer. Later, the stored reward r_t will be used to calculate the target Q-values using the Bellman equation, while s_{t+1} will be fed into the target network to compute Q_{t+1}.

2.2 The Algorithm

The algorithm for the DDPG method is expressed as follows. The algorithm provided is adapted from Lillicrap et al. (2016).

The DDPG Algorithm

```
Initialize:
(1) Randomly initialize the critic network $Q(s, a|\theta^Q)$ and the actor network $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$.
Initialize the replay buffer $R$.
for episode = 1 to $M$ do
    (2) Initialize a normal random process for action exploration.
    Receive the initial observation state $s_1$.
    for $t = 1$ to $T$ do
        (3) Select action $a_t = \mu(s_t|\theta^\mu) + N_t$, incorporating exploration noise.
        (4) Execute action $a_t$, observe reward $r_t$, and transition to the next state.
        (5) Store the transition $(s_t, a_t, r_t, s_{t+1})$ in the replay buffer $R$.
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$.
        (6) Set the target value:
            $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.
        (7) Update the critic by minimizing the loss:
            $L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$.
        (8) Update the actor policy using the policy gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i}$.
        (9) Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$,
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$.
    end for
end for
```

Most aspects of the algorithm have been discussed earlier; however, a few modifications are worth noting. Firstly, the weights of the actor and critic networks are randomly initialized. During each epoch, From the first to the last (M), the agent acts in the environment for time 1 to T. The action a_t calculated using the predictor actor network is then combined with a noise term to promote exploration, particularly in the initial stages of the training process. After executing the action a_t, the resulting experience tuple (s_t, a_t, r_t, s_{t+1}) is stored in a replay buffer. The replay buffer is divided into equal-sized batches to minimise biases during training, and data is randomly sampled from these batches.

The parameter y_i is the estimated Q-values calculated from the Bellman equation (1). The mean square of the differences between the estimated Q-value (y_i) and the predicted Q-value ($Q(s_t, \mu(s_t))$) is the loss function for this algorithm. Note that this loss is calculated over a random batch with the size of N. The target network can be updated using this loss function. Based on *the Deterministic Policy Gradient Theorem*, the formula for the actor policy update can be derived as shown in the algorithm. Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M. (2014). Finally, the target network is soft-copied of the predictor networks with the constant τ, which determines the strength of the copy.

3 Code Demonstration

3.1 The Environment Selection

The objective was to implement the DDPG method in a simple and visually intuitive environment to better observe the results. A suitable platform for this purpose is OpenAI Gym, which offers a variety of environments with different action spaces and levels of complexity. To practically demonstrate the fundamentals of the DDPG method, I selected the Classic Control Library. Since DDPG requires a continuous action space, the `Pendulum-v1` environment was chosen as the fitting option to implement the algorithm.

The action is the torque applied to the pendulum. The pendulum is released from a random uniform angle in the range of $[-\pi, \pi]$ and a random uniform velocity in the range of $[-1, 1]$. The goal of learning is to find a policy for applying torque that stabilizes the pendulum at the upright position ($\theta = 0$).

$$r = -\left(\theta^2 + 0.1\,\dot{\theta}^2 + 0.001\,\tau^2\right)$$

This equation encourages the agent to minimise the torque used, as the torque term has a negative coefficient and is squared. Consequently, the policy becomes more optimal by balancing efficiency and effectiveness. Furthermore, as the angle θ gets closer to zero, the reward increases. This guides the algorithm toward achieving the upright position for the pendulum while penalising deviations from the desired state.

The action space is continuous and has a size of one (the single torque applied to the pendulum). The state space has three components:

- $x = \cos(\theta)$
- $y = \sin(\theta)$
- The angular velocity of the pendulum $\dot{\theta}$

For more information, please visit the Gym Library website.

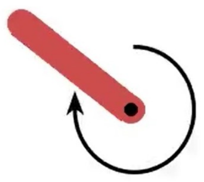


Figure 2: The Pendulum-v1 Environment on OpenAI Gym

3.2 The Code Explanation

I tried to implement the DDPG algorithm myself using the PyTorch library in Python. However, I have consulted with generative AI to write the code. Additionally, some syntaxes from Dr. Janan and his team's lab session code were used. The main text of the code can be found in my GitHub page. To be as concise as possible, the focus of this part is only on explaining the training portion of the code.

3.3 Hyper-parameters and Dimensions

Both the actor and critic networks have two hidden layers with dimensions of 512 neurones. As mentioned above, the input of the actor network is the state of the system which has three components. The input of the critic network are actions and the states, since the action is a single scalar torque, the input dimension of the critic network is $3 + 1 = 4$.

The learning rates for the actor and critic networks are 4×10^{-4} and 10^{-3} accordingly. The discount factor (γ) is 0.99. The batch size is chosen at 128 for effective training. Moreover, the stochastic Adam optimiser has been utilised for the training of the networks.

Network Definition and Initialization

```
state_size = 3  # [cos(theta), sin(theta), theta_dot]
action_size = 1  # Torque
num_hidden = 2

# Define networks
predictor_actor = MLP(state_size, action_size, num_hidden,
hidden_size=512)
predictor_critic = MLP(state_size + action_size, 1,
num_hidden, hidden_size=512)
initialize_weights(predictor_actor)
initialize_weights(predictor_critic)

target_actor = MLP(state_size, action_size,
num_hidden, hidden_size=512)
target_critic = MLP(state_size + action_size, 1,
num_hidden, hidden_size=512)
initialize_weights(target_actor)
initialize_weights(target_critic)

# Training parameters
max_epoche_number = 200
gamma = 0.99
batch_size = 128
max_buffer_size = 50000
max_action = 2.0  # Torque bounds in Pendulum-v1
critic_learning_rate = 1e-3
actor_learning_rate = 4e-4
actor_loss = torch.tensor(0.0)

optimiser_critic = optim.Adam(predictor_critic.parameters(),
lr=critic_learning_rate)
optimiser_actor = optim.Adam(predictor_actor.parameters(),
lr=actor_learning_rate)
```

3.4 The Training Loop

The following code snippet implements the core training loop of the DDPG algorithm. It involves sampling from the environment, storing experiences in a replay buffer, and using gradient updates to optimise both actor and critic networks. The order of the DDPG algorithm is shown with red numbers in Part 2.2. Ordinal numbers are also shown in the training loop code on the next page.

Main Loops: (3) (4) (5)

```
while epoch <= max_epoche_number:

    total_reward = 0
    state = env.reset()
    state[2] /= 10.0  # Normalize theta_dot
    done = False
    experience_counter = 0
    decay_factor = 0.998 ** epoch #Set up a decay factor for
        smoother updates in the long-term

    while not done:
```

(3)

```
        state_tensor = torch.tensor(state, dtype=torch.float32).
            view(1, -1)
        action = predictor_actor(state_tensor).detach().numpy()
        noise_scale = max(0.7 * decay_factor, 0.05)
        noisy_action = action + noise_scale * np.random.normal(0,
            0.2, size=action.shape)
        noisy_action = np.clip(noisy_action, -max_action,
            max_action)
```

(4)

```
        next_state, reward, done, truncated = env.step(
            noisy_action)
        done = done or truncated
        next_state[2] /= 10.0  # Normalize theta_dot
```

(5)

```
        # Store experience in the replay buffer
        state_list.append(state)
        action_list.append(noisy_action)
        reward_list.append(reward)
        next_state_list.append(next_state)
        total_reward += reward
        experience_counter += 1
```

A decay factor is established with the formula 0.998^{epoch} to ensure that the learning rate decreases as the training process progresses, leading to more stable steady-state responses. Furthermore, the third component of the state, $\dot{\theta}$, is normalised by dividing it by a factor of 10. Since the first two components of the state are $\sin(\theta)$ and $\cos(\theta)$, which are inherently bounded between $[-1, 1]$, normalising $\dot{\theta}$ ensures that all state variables are on a comparable scale. This normalisation facilitates more efficient network training.

A normal noise distribution, $\mathcal{N}(0, 0.2^2)$, has been added to the actions to encourage exploration of the environment. The amplitude of the noise goes to zero as the decay factor decreases over epochs, which stabilises the final states of the training by favouring exploitation in the steady-state phase.

Update Networks and Soft-Copy (6) (7) (8)

```
# Train if buffer has enough samples
        if len(state_list) >= batch_size:
            batched_state, batched_action, batched_reward,
                batched_next_state = batch_data(
                 state_list, action_list, reward_list,
                     next_state_list, batch_size)
```

(6)

```
target_miu = target_actor(batched_next_state)
target_q = target_critic(torch.cat([batched_next_state
    , target_miu], dim=1)).detach()
y = batched_reward + gamma * target_q
```

(7)

```
# Update critic
optimiser_critic.zero_grad()
critic_loss = MSE_loss(predictor_critic(torch.cat([
    batched_state, batched_action], dim=1)), y)
critic_loss.backward()


optimiser_critic.step()
```

(8)

```
# Update actor
optimiser_actor.zero_grad()
actions_pred = predictor_actor(batched_state)
q_values = predictor_critic(torch.cat([batched_state,
    actions_pred], dim=1))
actor_loss = -q_values.mean()

for param_group in optimiser_actor.param_groups:
     param_group['lr'] = -actor_learning_rate if
         actor_loss.item() < 0 else actor_learning_rate

actor_loss.backward()


optimiser_actor.step()
```

(9)

```
# Update target networks
soft_update(target_actor, predictor_actor,
tau = 0.005)
soft_update(target_critic, predictor_critic,
tau = 0.005)
```

When the actor network loss turns negative, the training process becomes unstable. Therefore, in step (8) an if function is added to the code to flip the sign of the learning rate in case of negative actor loss. Two modules for Bach data structure and soft-copy are presented on the next page in green boxes. These modules are usually defined before the training loop; however, they are mentioned after it to avoid confusion.

Batch Data Selection Module

```
def batch_data(state_list, action_list, reward_list,
    next_state_list, batch_size):
    num_samples = len(state_list)
    sample_indices = random.sample(range(num_samples), batch_size)


    # Ensure tensors are correctly reshaped
    batched_state = torch.cat(
        [torch.tensor(state_list[i], dtype=torch.float32).view(1,
            -1) for i in sample_indices], dim=0
    )
    batched_action = torch.cat(
        [torch.tensor(action_list[i], dtype=torch.float32).view(1,
            -1) for i in sample_indices], dim=0
    )
    batched_reward = torch.cat(
        [torch.tensor([reward_list[i]], dtype=torch.float32).view
            (1, -1) for i in sample_indices], dim=0
    )
    batched_next_state = torch.cat(
        [torch.tensor(next_state_list[i], dtype=torch.float32).
            view(1, -1) for i in sample_indices], dim=0
    )

    return batched_state, batched_action, batched_reward,
        batched_next_state
```

This module is designed to create a layered data structure for the replay buffer with a size of 128. During each training iteration, a random batch is selected from the created layers and fed into the training loops. This stochastic approach reduces biases and improves the overall training process.

Soft-Copy and the Mean Square Error (MSE) Modules

```
def MSE_loss(prediction, target):

      return ((prediction - target)**2).sum(dim=-1).mean()

def soft_update(target_network, predictor_network, tau):
        """
        Softly updates the parameters of the target network.
        target_param = tau * predictor_param + (1 - tau) *
            target_param
        """
        for target_param, predictor_param in zip(target_network.
            parameters(), predictor_network.parameters()):
            target_param.data.copy_(tau * predictor_param.data +
                (1.0 - tau) * target_param.data)
```

4 Results and Visualisations

Training a neural network is a complex task that requires extensive hyperparameter tuning. Due to the inherent instability issues associated with the Deep Deterministic Policy Gradient (DDPG) method, training such models demands significant effort and experimentation. Although the training outcomes in this case are not perfectly optimal, they demonstrate satisfactory quality for tutorial purposes and will be presented in the following section.

4.1 Learning Curves

The code was executed over 200 epochs, and Figure 3 shows the losses for the predictor actor and predictor critic networks. The x-axis represents the total number of code iterations. Since there were 200 epochs and in each epoch, the time limit for simulation was 200, the total number of interactions should be $200 \times 200 = 40,000$. The y-axis represents the network losses calculated within steps (7) and (8) of Part 3.4.

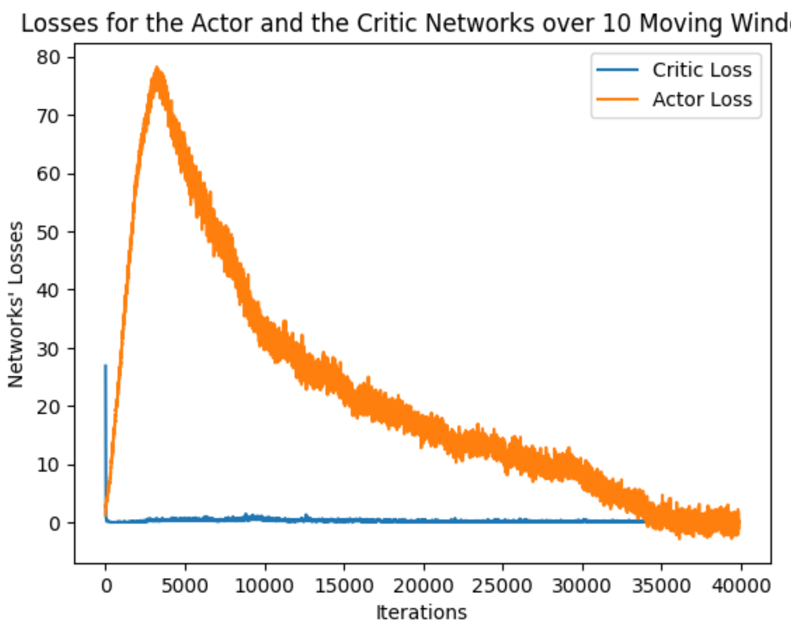


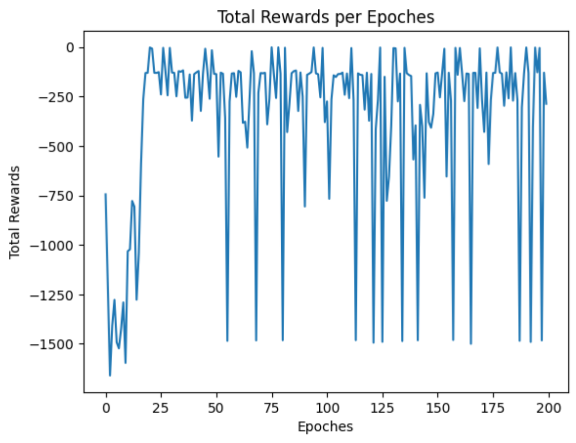
Figure 3: Predictor Actor and Critic Networks Losses Over Total Iterations

The critic network converged promptly within the first epochs of the simulation; however, the actor network's losses converged relatively slowly and exhibited more fluctuations compared to the critic network. The training of these networks is highly sensitive to changes in hyperparameters, and tuning the networks to ensure convergence is a challenging task.

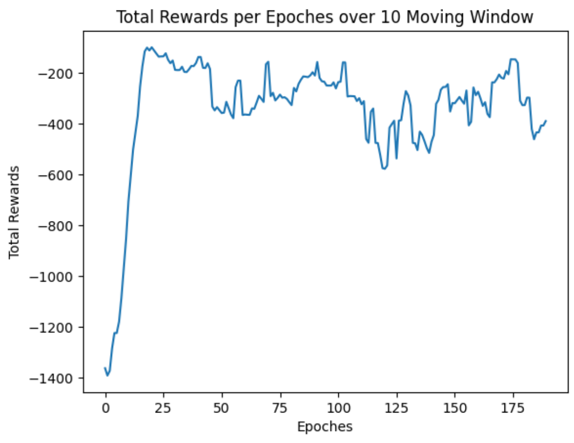
The actor network's average loss approaches zero in the steady-state phase but remains fluctuating. By reducing the effect of noise, the fluctuations in the actor network's losses are reduced, but the system does not explore the environment effectively, resulting in suboptimal final rewards. Thus, there is a trade-off between stability and exploration, which complicates the tuning process.

Each of the 200 epochs represents one complete simulation in the environment. Therefore, the total rewards collected in each simulation correspond to the total rewards collected in each epoch. Ideally, as the system training progresses, higher rewards should be achieved, as the essence of Reinforcement Learning is to maximise rewards.

Figure 4 (a) illustrates the average rewards obtained over 200 epochs. The total rewards increase as the networks are progressively trained. However, fluctuations in rewards are observed because of the noise incorporated during action exploration. As mentioned earlier, there exists a trade-off between stability and exploration. Reducing noise can improve stability, but hamper exploration, which is crucial to achieving optimal rewards. Balancing this trade-off is a fundamental challenge in Reinforcement Learning.



(a) Total rewards



(b) Total rewards with moving 10 window

Figure 4: Total Rewards per Epoches

In order to better interpret this figure, a moving average with a window size of 10 was applied. This means that the average of every 10 consecutive rewards was calculated and this value was propagated throughout the data. This approach results in Figure 4(b), which shows reduced fluctuations and therefore provides a clearer and more interpretable representation of the rewards. As can be seen in Figure 4 (b), total rewards have increased during the training process, which is in line with our expectations.

4.2 Phase Diagram

The Phase Diagram is an insightful approach to interpreting the results of the simulation. The x-axis of the phase diagram represents the pendulum's angle, while the y-axis represents the pendulum's angular velocity. This diagram is implicitly a function of time and its goal is to illustrate the trajectory of the pendulum during a single simulation.

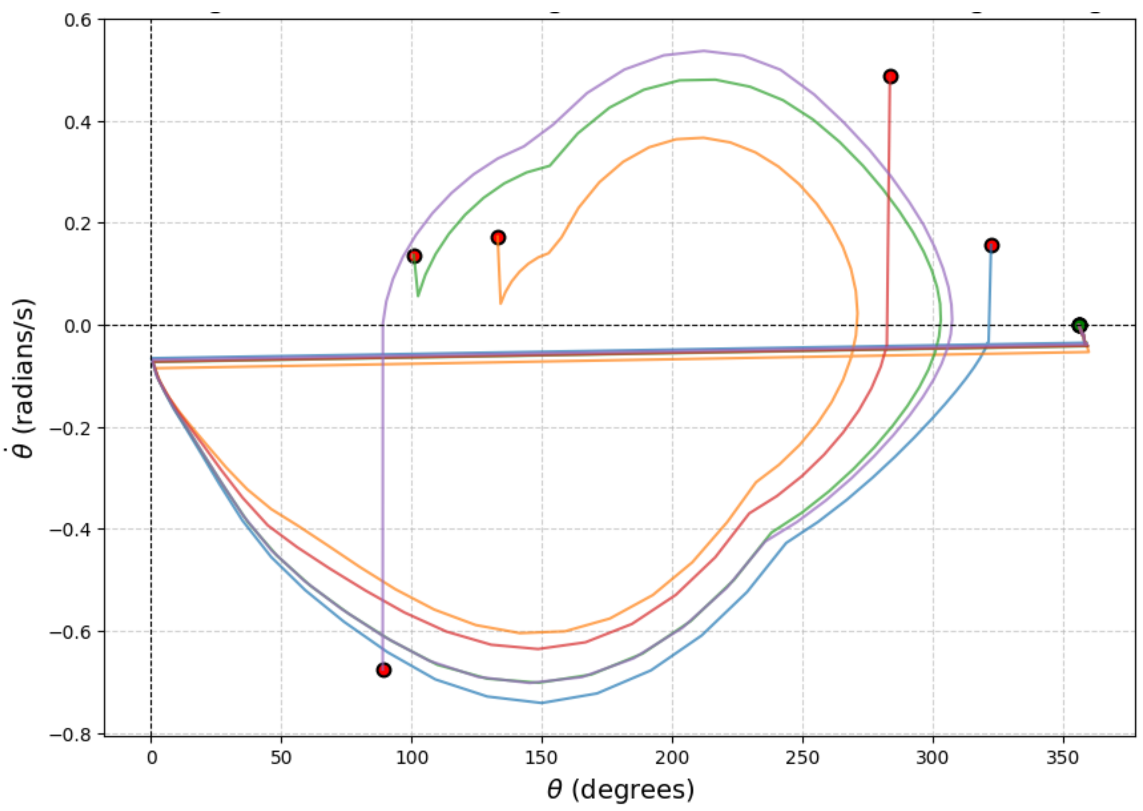


Figure 5: The Phase Diagram of the Pendulum

Initially, the pendulum is released from a random uniform angle and a random uniform initial angular velocity. Therefore, the initial state of the pendulum is a random uniform point in the phase plane. These initial points are indicated by red dots. Over time, the angle and angular velocity of the pendulum change as it moves. The goal of training is to stabilize the pendulum in the upright position (zero angle) with zero velocity. Ideally, in a perfectly trained system, the trajectories should converge to a zero angle and zero angular velocity. These final states are marked by green dots.

In conclusion, each trajectory represents one simulation of the pendulum, beginning with a red dot (initial state) and ending with a green dot (final state). Figure 5 shows the phase diagram for five runs (trajectories). Since the training is not perfect, the green dots do not align precisely with zero angle and zero angular velocity, resulting in an offset in the final angles. This offset can be interpreted as the steady-state error in controlling the pendulum. Additionally, the states appear to jump from zero angle to 360 in Figure 5. This is reasonable because of the wrapped behaviour of the angle.

One notable observation is the vertical lines that originate from the red dots. These occur because of the immediate torque applied to the pendulum after its release from a random state. From dynamics, the relationship between the angular acceleration and the torque is given by $T = I\alpha$. At the beginning of the simulation, the torque starts at zero, creating an impulse in the system. This impulse causes an immediate change in the angular acceleration. Since angular acceleration is the gradient of angular velocity, it also induces an impulse in angular velocity, resulting in the vertical lines in the phase diagram.

Following the initial impulses, all trajectories exhibit a similar pattern and eventually converge to their respective green dots. This consistency is fascinating, as it suggests that the network attempts to replicate a predictable and consistent shape in the phase plane during the training process.

4.3 Histogram

Another simple yet insightful way to evaluate the outcomes of training is through a histogram representation of the data. Figure 6 illustrates the histogram of final angles over 100 runs. In an ideal scenario, where training is flawless, the final angles would cluster tightly around zero. However, achieving an exact angle of zero is highly improbable due to the inherent complexity of the system and the limitations of training. Consequently, the histogram reveals a distribution centred around the zero angle. Despite this, the majority of the runs fall within a close range of zero, demonstrating that the training achieved an acceptable level of quality. This distribution highlights the system's ability to consistently bring the pendulum toward the desired upright position, albeit with slight deviations caused by noise or imperfections in the training process. Two distant portions of the distribution on both sides are not outliers. Same as the last section, due to the wrapping behaviour of the angle there is a jump in the diagram, yet not an actual flaw of the training.

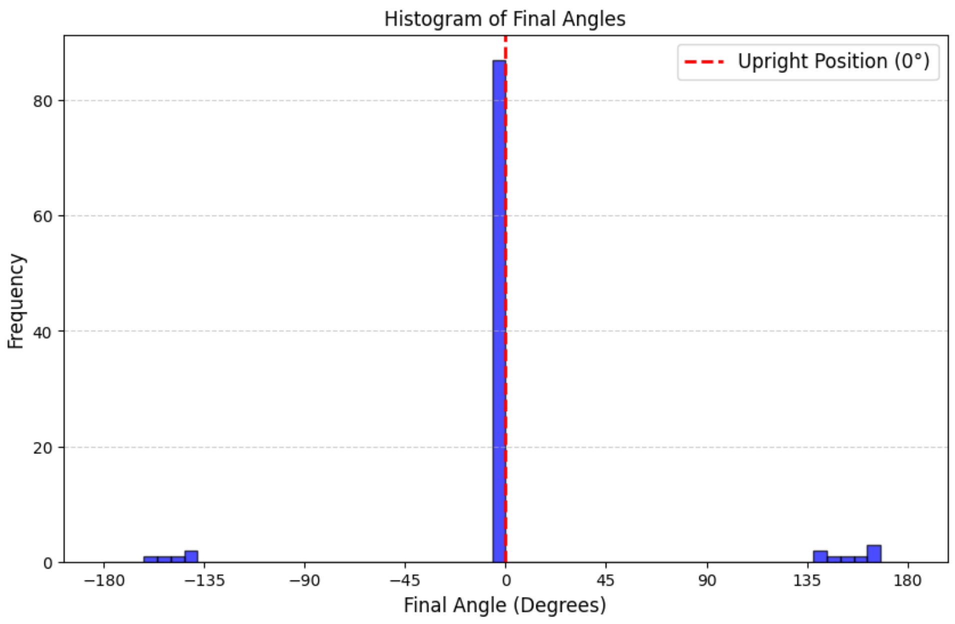


Figure 6: Histogram of the Final Angles Over 100 Runs

4.4 Plotting $\theta = f(t)$

Another beneficial figure that can be used to inspect the outcomes of the training is the plotting of the variation of angles over time. Figure 7 shows the figure mentioned above, where the y-axis is the angle θ and the x-axis is the time t. The initial angles are generated randomly; therefore, at time $t = 0$ all the angles are random points on the y-axis, that are shown with red dots.

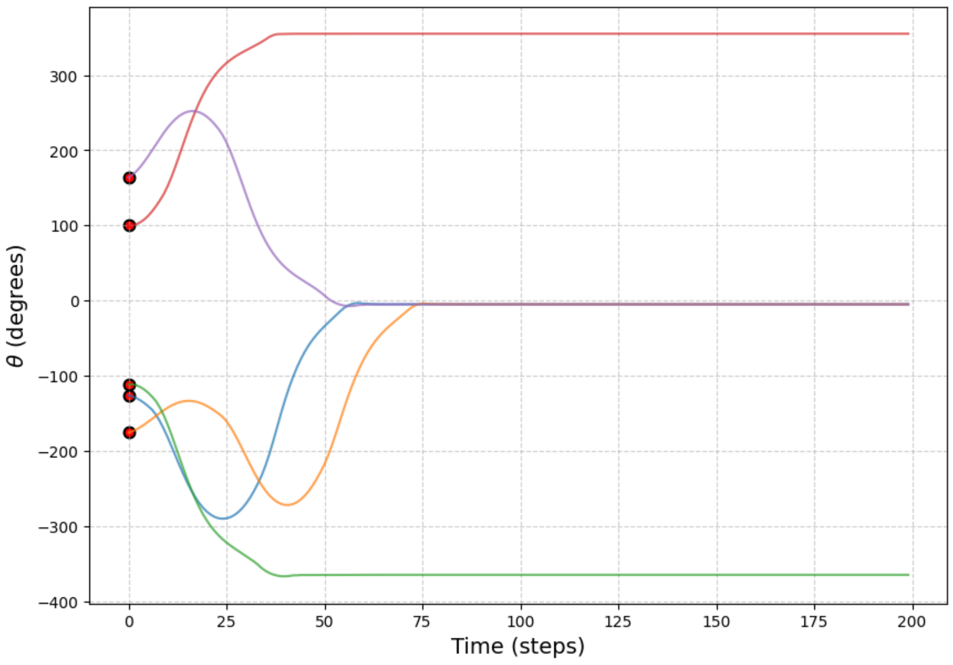


Figure 7: The diagram of angles over time

The maximum bound of the time is 200, which is the default maximum simulation time of the Pendulum-v1 environment. The angles converged to zero (or 360 equivalently) before 75 time units. This indication demonstrates the speed of the system that can converge fast.

5 Conclusion

This tutorial aimed to introduce the Deep Deterministic Policy Gradient (DDPG) algorithm in reinforcement learning. By following this tutorial, you should now have a comprehensive understanding of the core logic behind the method, its implementable algorithm, and a practical implementation using PyTorch in the Pendulum-v1 environment.

The implemented code demonstrates some acceptable results; however, it is not flawless. The model exhibits a steady-state error and the training process lacks robustness, meaning that the results can vary between different executions. The DDPG method is a powerful approach for continuous control tasks in real-world applications, benefiting from its model-free nature and its ability to handle high-dimensional action spaces (Lillicrap et al., 2016). Furthermore, it is relatively easier to implement compared to some modern deep reinforcement learning algorithms.

However, the DDPG method is prone to instability and requires considerable effort to adjust and stabilise the hyperparameters (Lillicrap et al., 2016). Modern reinforcement learning methods such as Advantage Actor-Critic (A3C), Asynchronous Advantage Actor-Critic (A2C), and Soft Actor-Critic (SAC) have emerged to address some of these challenges by enhancing stability, improving sample efficiency, and reducing the need for hyperparameter tuning (Haarnoja et al., 2018).

6 Acknowledgment

The author acknowledges the use of ChatGPT-4.0 (OpenAI) for assistance in various aspects of this project. Specifically, ChatGPT provided support in debugging and optimising the Python code, refining the structure and content of the report, and improving the clarity of language, grammar, and technical explanations. ChatGPT also offered guidance on formatting LaTeX elements, structuring visualisations, and preparing documentation for the project's GitHub repository. The author confirms that AI-generated content has not been presented as original work. All contributions from ChatGPT were supplementary to the author's own efforts in analysis, design, and report preparation.

7 Bibliography

- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D. (2016). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971.*

- Haarnoja, T., Zhou, A., Abbeel, P., Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv preprint arXiv:1801.01290.*

- Khamies, W. D., Mohammedalamen, M., Rosman, B. (2024). Transfer Learning for Prosthetics Using Imitation Learning. In *Proceedings of the International Conference on Robotics and Automation (ICRA).*

- Kumar, A., Paul, N., Omkar, S. N. (2024). Bipedal Walking Robot using Deep Deterministic Policy Gradient. In *Proceedings of the International Conference on Robotics and Automation (ICRA).*

- Sutton, R. S., Barto, A. G. (1998). Reinforcement Learning: An Introduction. MIT Press.

- OpenAI. (2024). ChatGPT-4.0: The AI Writing Assistant. Available at: `https://chat.openai.com/` [Accessed 10 Dec. 2024].