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

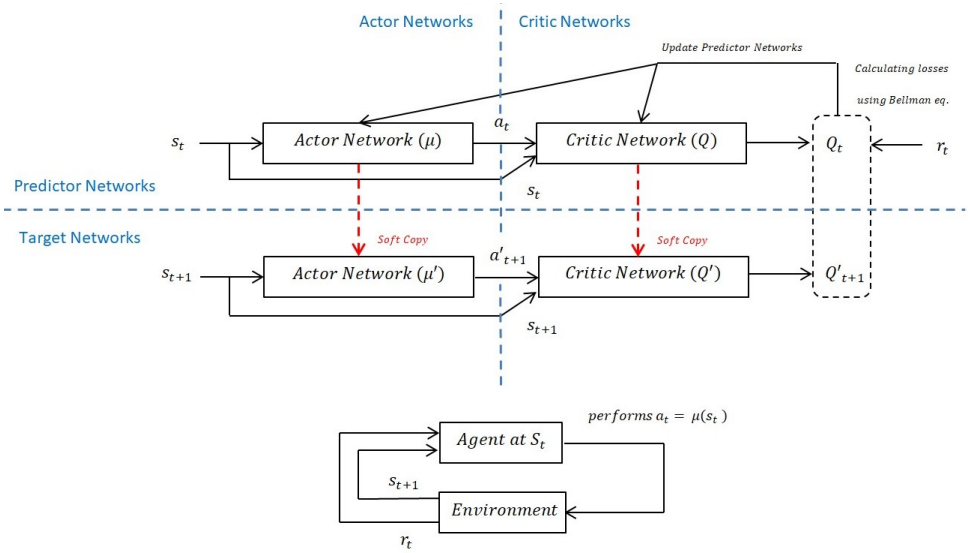
Alireza Khaleghi

Reinforcement Learning for Bioengineers

Department of Bioengineering

Imperial College London

Autumn 2024



1 Introduction

Reinforcement learning (RL) is a machine learning approach where agents learn to make decisions by interacting with an environment to maximise cumulative rewards. Among RL methods, the **Deep Deterministic Policy Gradient (DDPG)** stands out as a powerful tool for continuous control tasks. It addresses the challenge of selecting precise actions in high-dimensional action spaces, making it widely used in robotics, control systems, and automation.

Key Advantages of DDPG:

- **Continuous Control:** DDPG can handle continuous action spaces, unlike Q-learning, which is limited to discrete actions.
- **Model-Free Approach:** It does not require a model of the environment, making it suitable for real-world tasks where models are unavailable.
- **Actor-Critic Framework:** The method combines the benefits of policy-based and value-based approaches, leading to more stable training.

Challenges Addressed by DDPG:

- **High-Dimensional Spaces:** Traditional RL struggles in environments with continuous, high-dimensional action spaces. DDPG efficiently handles these spaces using neural networks.
- **Stability of Learning:** Reinforcement learning methods can be unstable due to the constantly changing policy. DDPG overcomes this with *soft-updated target networks*, which ensure smoother updates.
- **Exploration vs Exploitation:** Exploration is crucial for learning effective policies, but excessive exploration leads to instability. DDPG addresses this by adding controlled noise to actions, promoting balance.

What is DDPG?

- **Deep:**
  - Uses **deep neural networks** to approximate the *actor* and *critic* functions.
  - The **actor network** maps states to actions, while the **critic network** estimates Q-values for state-action pairs.
  - Enables handling of high-dimensional, complex state, and action spaces.
- **Deterministic:**
  - Unlike stochastic policies, the DDPG selects a **specific action** for a given state.
  - This contrasts with stochastic policies that produce a probability distribution over possible actions.
  - Deterministic policies simplify action selection, making the method suitable for continuous control tasks.

- **Policy Gradient:**
  - The actor network is updated using the **policy gradient** methods, which adjust the weights of the network to maximise expected rewards.
  - DDPG relies on the *Deterministic Policy Gradient Theorem*, which enables policy updates using gradients of Q values.
  - Unlike value-based methods (e.g., Q-learning), policy gradients directly optimise the action-selection process.

DDPG combines two key components:

- **Actor Network:** Responsible for selecting continuous actions based on the current state.
- **Critic Network:** Estimates Q-values for state-action pairs and guides the actor's policy updates.

The actor's output action is evaluated by the critic to compute errors for network updates. To ensure stability, DDPG employs *target actor* and *target critic* networks, which are **soft-updated** copies of the predictor networks. This design significantly improves learning stability and convergence, enabling DDPG to solve complex continuous control problems.

1.1 DDPG in Bioengineering

DDPG is widely used in robotics and control, notably in bioengineering for personalised prosthetics and bipedal robots. Prosthetics face the challenge of adapting to individual gait patterns. Using DDPG's model-free learning, prosthetics can personalise and improve performance (Khamies et al., 2024). Similarly, bipedal robots struggle to mimic the human gait cycle. DDPG enables robots to stabilise their gait and achieve more human-like walking behaviour (Kumar et al., 2024).

2 Theoretical Explanation

2.1 Overview of Method and Notations

This section is based on the work of Lillicrap et al. (2016) on Continuous Control with Deep Reinforcement Learning.

The Bellman equation for Temporal Difference (TD) learning estimates Q-values, which are used to calculate the error between predicted and actual Q-values generated by the critic network:

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

Here, $Q(s_t, a_t)$ is the Q-value for action a_t in state s_t at time t; $r(s_t, a_t)$ is the reward for

a_t in s_t; γ is the discount factor; and $Q(s_{t+1}, a_t)$ is the Q-value for the next state s_{t+1}, estimated using the target network.

The actor and critic networks are represented as μ and Q, respectively, with their weights indicated as θ^μ and θ^Q. The stability target networks are marked as Q' and μ'. The actor network takes the state s_t as input and outputs a continuous action a_t:

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

This action a_t is then combined with s_t and fed into the critic network to calculate the Q-value:

$$Q(s_t, a_t|\theta^Q) = Q(s_t, \mu(s_t)|\theta^Q) \tag{3}$$

Figure 1 illustrates the general structure of the DDPG method.

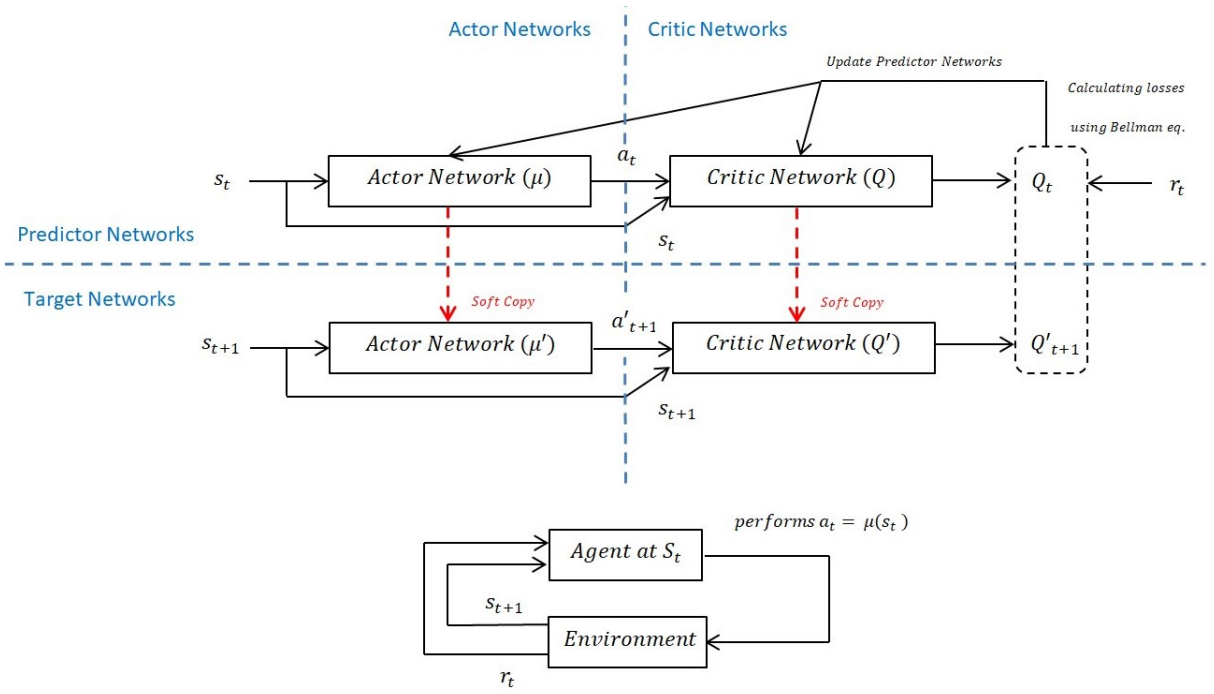


Figure 1: The Schematic of the DDPG Method

As shown in Figure 1, the network of predictor actors selects the action a_t, which the agent performs at time t in the environment. This results in a reward r_t and a transition to the next state s_{t+1}. The interaction is stored as an experience tuple (s_t, a_t, r_t, s_{t+1}) in a replay buffer. The stored reward r_t is used to calculate the target Q values using the Bellman equation, while s_{t+1} is fed into the target network to calculate 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 θ^Q and θ^μ.
Initialise target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$.
Initialise 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 minimising 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 of the key aspects of the algorithm have been discussed, but some notable modifications remain. The weights of the actor and critic networks are initialised randomly. During each epoch, from 1 to M, the agent interacts with the environment for T time steps. The action a_t, generated by the network of predictor actors, is combined with a noise term to encourage exploration, especially during early training. After executing a_t, the resulting experience tuple (s_t, a_t, r_t, s_{t+1}) is stored in a replay buffer. This buffer is divided into mini-batches of size N, which are randomly sampled to reduce training bias.

The Q-value target y_i is calculated using the Bellman equation (1). The loss is computed as the mean squared error (MSE) between the estimated Q value (y_i) and the predicted Q value of the critic ($Q(s_t, \mu(s_t))$). This loss function updates the target network. Actor policy updates are based on the *Deterministic Policy Gradient Theorem* (Silver et al., 2014). Finally, the target networks are soft-updated from the predictor networks using a soft-update factor τ, which controls the update rate.

3 Code Demonstration

3.1 Environment Selection

To demonstrate the DDPG method, a simple and visually intuitive environment was chosen to effectively observe the results. OpenAI Gym, which offers various environments with diverse action spaces, was selected for this purpose. Among its libraries, the `Classic Control` Library was deemed appropriate due to its simplicity. Since DDPG requires a continuous action space, the `Pendulum-v1` environment was chosen as an ideal candidate for implementation.

In this environment, the action corresponds to the torque applied to the pendulum. The pendulum starts from a random uniform angle within $[-\pi, \pi]$ and a random uniform angular velocity within $[-1, 1]$. The objective is to find a policy for torque application that stabilises the pendulum in an upright position ($\theta = 0$).

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

The reward function encourages minimal torque usage since the torque term is squared and negatively weighted. Consequently, an optimal policy strikes a balance between torque efficiency and stabilisation. Furthermore, as θ approaches zero, the reward increases, guiding the algorithm to maintain the pendulum upright while minimising deviations.

The action space is continuous with a single action (torque). The state space consists of three components:

- $x = \cos(\theta)$
- $y = \sin(\theta)$
- Angular velocity $\dot{\theta}$

For more details, please visit the Gym Library website.

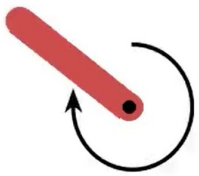


Figure 2: Pendulum-v1 Environment on OpenAI Gym

3.2 Code Explanation

The DDPG algorithm was implemented using PyTorch, with the support of generative AI for code optimisation. Some syntax from Dr. Janan's lab session was also incorporated. The complete code can be found on my GitHub page. The focus of this section is to explain the key aspects of the training process.

3.3 Hyperparameters and Network Dimensions

The actor and critic networks have two hidden layers, each with 512 neurones. The actor network takes a 3-dimensional state input and outputs a continuous action. The critic network has an input size of 4 (3 for the state and 1 for the action). The key hyperparameters and network configurations are summarised in Table 1.

Table 1: Key Hyperparameters and Network Configurations

Parameter	**Value**
Hidden Layer Size	512 neurons (2 layers)
Actor Input Size	3 (state components)
Critic Input Size	4 (state + action)
Actor Learning Rate	4×10^{-4}
Critic Learning Rate	1×10^{-3}
Discount Factor (γ)	0.99
Batch Size	128
Optimiser	Adam

3.4 The Training Loop

The core training loop of the DDPG algorithm involves sampling from the environment, storing experiences in a replay buffer, and using gradient updates to optimise the actor and critic networks. The steps of the DDPG algorithm are marked with red ordinal numbers corresponding to the steps in Part 2.2.

Training Loop: (3) (4) (5)

```
while epoch <= max_epoche_number:

    total_reward = 0
    state = env.reset()
    state[2] /= 10.0  # Normalize theta_dot
    done = False
    decay_factor = 0.998 ** epoch  # Decay factor for smoother
        updates

    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
```

The training loop follows a structured process where each epoch represents an iteration of training. The key concepts within this loop are as follows.

- **Decay Factor:** A decay factor (0.998^{epoch}) is used to reduce the amplitude of noise over epochs. This facilitates exploration at the beginning and exploitation at later stages, improving the stability of the final results.
- **Normalization:** The third component of the state, $\dot{\theta}$, is normalised by dividing it by 10. The first two components $\sin(\theta)$ and $\cos(\theta)$ are naturally bounded between $[-1, 1]$, but $\dot{\theta}$ is not. Normalising ensures that all state components have a similar range, improving the efficiency of network training.

- **Exploration Noise:** The action is perturbed by normal noise $\mathcal{N}(0, 0.2^2)$ to promote exploration. The amplitude of the noise decreases over time as the decay factor reduces, leading to more exploitation in later epochs.
- **Replay Buffer:** Each interaction with the environment, i.e., (s_t, a_t, r_t, s_{t+1}), is stored in a replay buffer for later training. This technique prevents the algorithm from overfitting to sequential correlations and improves training efficiency.

Update Networks and Soft-Copy (6) (7) (8) (9)

```
# 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)

```
# (6) Calculate target Q-values
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)

```
# (7) Update critic network
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)

```
# (8) Update actor network
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()

# Update actor learning rate conditionally
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)

```
# (9) Soft update of target networks
soft_update(target_actor, predictor_actor, tau=0.005)
soft_update(target_critic, predictor_critic, tau=0.005)
```

4 Results and Visualisations

Training neural networks, especially with the DDPG method, requires careful hyperparameter tuning to address its inherent instability. Although the training process may not yield optimal results, the outcomes presented here demonstrate satisfactory performance for tutorial purposes.

4.1 Learning Curves

The model was trained for 200 epochs, with each epoch involving 200 simulation steps, resulting in a total of $200 \times 200 = 40,000$ iterations. Figure 3 shows the losses of the predictor actor and the critic network during these iterations. The x-axis represents the total iterations, while the y-axis denotes the loss calculated during Steps (7) and (8) of the DDPG algorithm.

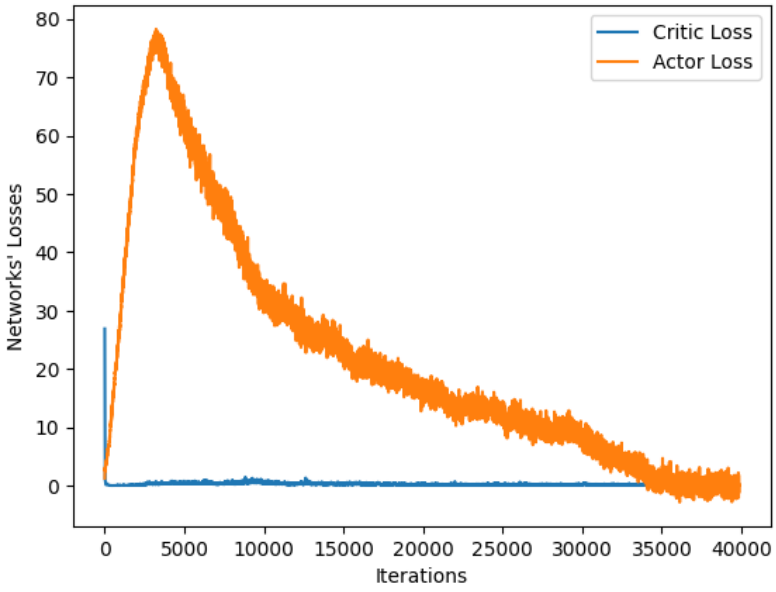


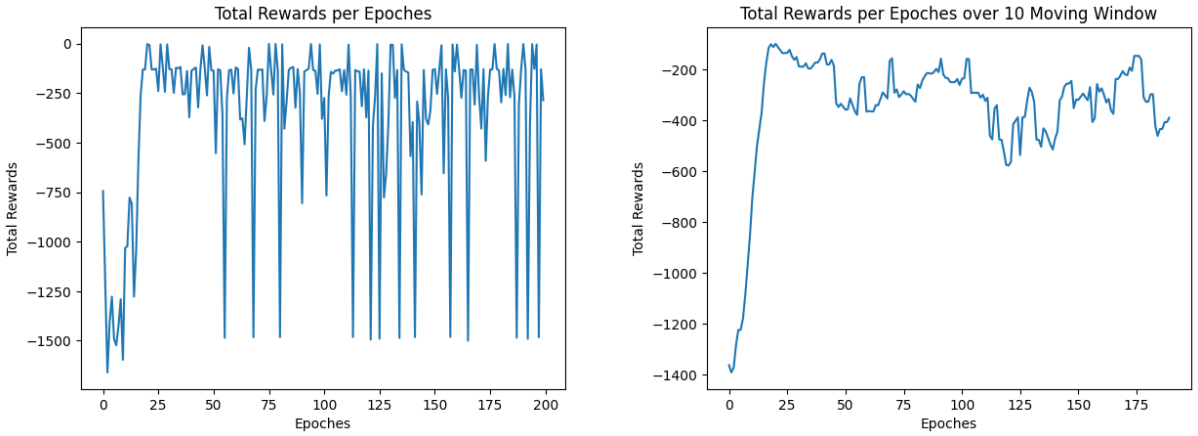
Figure 3: Losses of Predictor Actor and Critic Networks Over Total Iterations

The critic network converged rapidly in the early stages, whereas the actor network's loss decreased more gradually and exhibited higher fluctuations. These fluctuations arise from the exploratory nature of the learning process, particularly in the actor's policy updates. Hyperparameter tuning plays an important role in ensuring stable training and convergence for both networks.

The actor network's average loss approaches zero during steady-state phases, but continues to fluctuate due to the balance between exploration and exploitation. Reducing exploration noise can stabilise training but may hinder the system's ability to explore effectively, resulting in suboptimal final rewards. This highlights the essential trade-off between exploration and stability, a fundamental challenge in reinforcement learning.

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 system training progresses, higher rewards should be achieved, as the essence of Reinforcement Learning is to maximise rewards.

Figure 4 (a) shows the average rewards for over 200 epochs. As training progresses, the total rewards increase, indicating improved policy performance. Fluctuations in rewards arise from the exploration noise added to the actions. As discussed earlier, there is a trade-off between exploration and stability. Reducing noise improves stability but limits exploration, affecting the agent's ability to discover better policies. Balancing this trade-off is a key challenge in reinforcement learning.



(a) Total rewards

(b) Total rewards with 10-step moving average

Figure 4: Total rewards over epochs

To better visualise trends, a 10-step moving average was applied in Figure 4(b). This smooths out fluctuations by averaging every 10 consecutive reward values. As seen, the moving average provides a clearer trend, revealing the increase in total rewards as training progresses. The upward trend in Figure 4(b) aligns with expectations, demonstrating that the agent learns to optimise its actions over time.

4.2 Phase Diagram

The phase diagram provides a clear view of the pendulum's dynamics during the simulation. The x-axis represents the pendulum's angle, while the y-axis shows its angular velocity. This diagram reveals the trajectory of the pendulum during a single simulation.

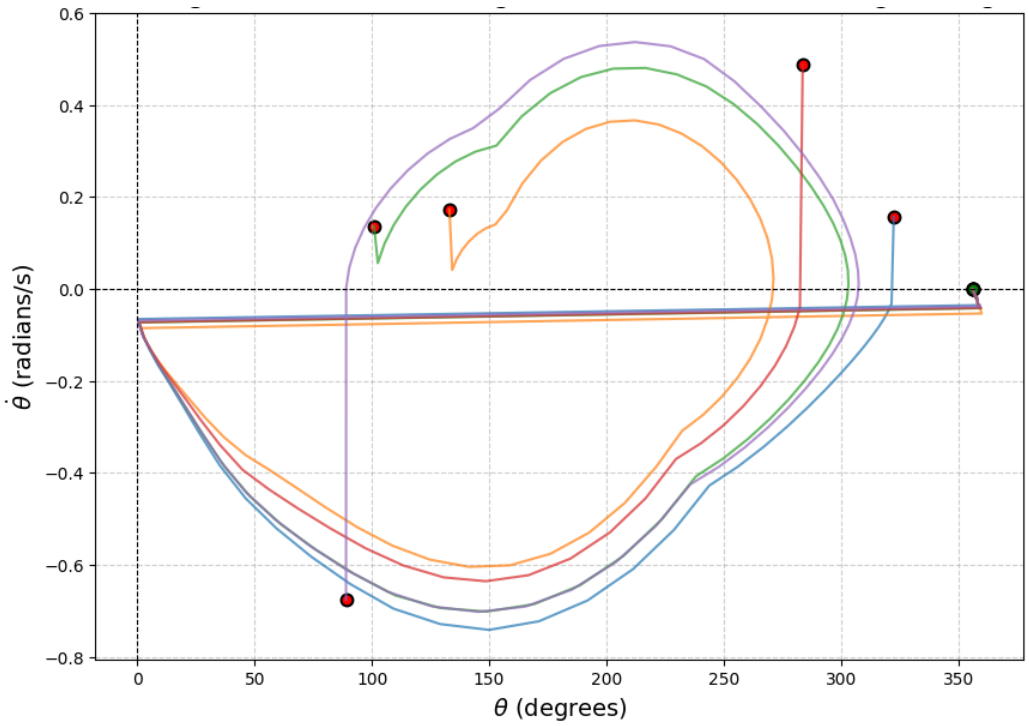


Figure 5: Phase diagram of the pendulum

At the start, the pendulum is released from a random angle and velocity, resulting in a random starting point in the phase plane, marked by red dots. Over time, the angle and velocity of the pendulum change as it moves. The goal of training is to stabilise the pendulum in an upright position (zero angle) with zero velocity. Ideally, each trajectory should end at (0, 0) in the phase space, but due to imperfect training, the final points (green dots) exhibit small deviations, interpreted as steady-state errors. The wrap-around effect from 0 to 360 degrees is visible, reflecting the cyclic nature of the angle.

A notable feature of the diagram is the vertical lines that emerge from the red dots. These lines result from an initial torque impulse applied at the beginning of the simulation. From the dynamics, torque (τ) is related to angular acceleration (α) as $\tau = I\alpha$, where I is the moment of inertia. The impulse causes an abrupt change in the angular acceleration, which, as a derivative of the velocity, produces an immediate change in the angular velocity.

After this initial impulse, the trajectories follow a consistent pattern, converging toward the green dots. This pattern suggests that the neural network attempts to produce consistent behaviour during the training process.

4.3 Histogram

The histogram provides a simple, yet effective way to assess training outcomes. Figure 6 shows the distribution of the final angles over 100 runs. Ideally, all final angles would cluster around zero, indicating perfect training. However, due to system complexity and training limitations, the final angles follow a distribution centred around zero. Most runs fall within a small range close to zero, demonstrating satisfactory training performance. The distant sections on both sides of the histogram are not outliers but result from the natural wrap-around behavior of the angle from 0 to 360 degrees.

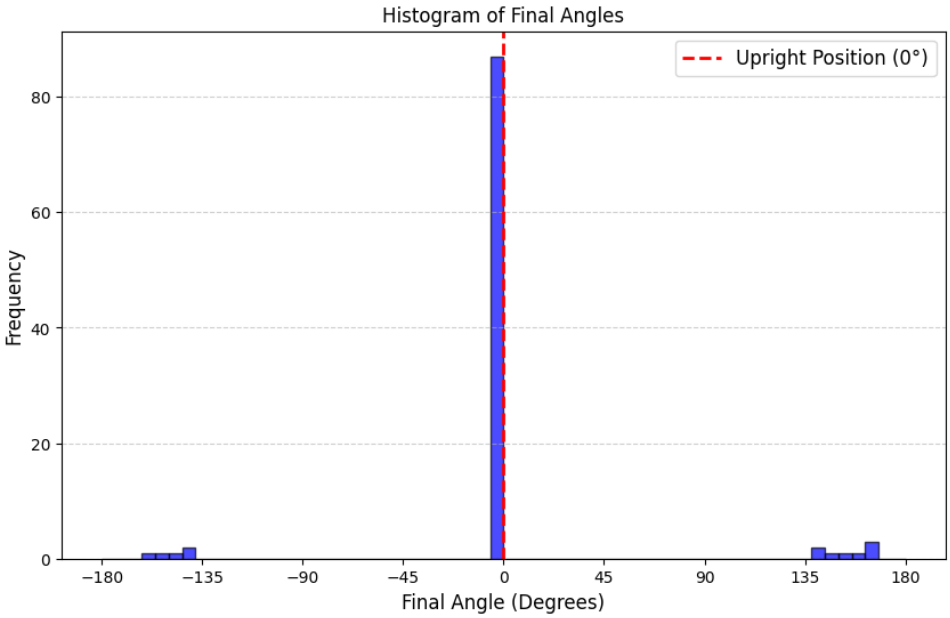


Figure 6: Histogram of the Final Angles Over 100 Runs

4.4 Angle Over Time $\theta = f(t)$

Figure 7 shows how the angle of the pendulum θ evolves over time t. The y-axis represents the angle, while the x-axis represents time. Randomly generated initial angles result in scattered starting points at $t = 0$ (marked as red dots).

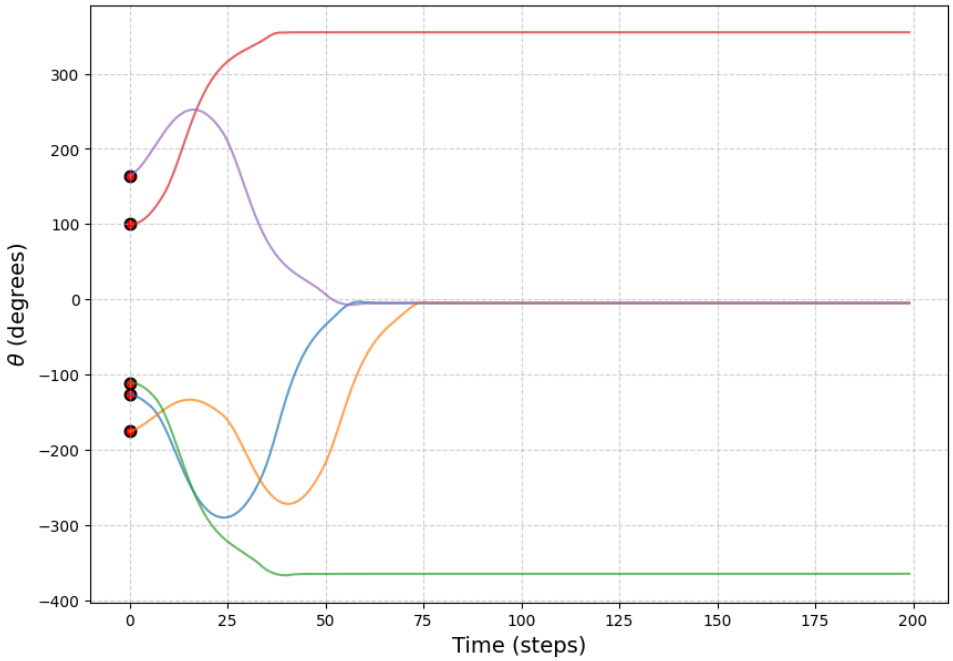


Figure 7: Angle Over Time for Multiple Runs

The simulation runs for a maximum of 200 time units, as set by the Pendulum-v1 environment. As shown, angles converge to zero (or 360 degrees due to angle wrapping) before 75 time units, highlighting the system's rapid convergence. The consistent convergence across multiple runs demonstrates the effectiveness and robustness of the DDPG algorithm for pendulum control.

5 Conclusion

This tutorial introduced the Deep Deterministic Policy Gradient (DDPG) algorithm for reinforcement learning. It provided an overview of the method's core logic, the step-by-step implementation, and a practical application using PyTorch in the Pendulum-v1 environment.

The implemented code achieved reasonable performance but exhibited steady-state error and variability across different runs. Despite these limitations, DDPG remains a strong choice for continuous control tasks due to its model-free nature and the capacity to manage high-dimensional action spaces (Lillicrap et al., 2016). It is also more straightforward to implement compared to more advanced deep-reinforcement learning algorithms.

However, DDPG is prone to instability and requires significant hyperparameter tuning (Lillicrap et al., 2016). Recent methods like Advantage Actor-Critic (A3C), Asynchronous Advantage Actor-Critic (A2C), and Soft Actor-Critic (SAC) have been developed to address these limitations by improving stability, increasing sample efficiency, and reducing the complexity of hyperparameter tuning (Haarnoja et al., 2018).

6 Acknowledgment

The author acknowledges the use of ChatGPT-4.0 (OpenAI) for support in debugging and optimising Python code, refining the report structure, and enhancing the clarity of technical explanations. ChatGPT also assisted with formatting LaTeX elements, visualisation structuring, and project documentation for GitHub. The author affirms that AI-generated content is not presented as original work. The contributions of ChatGPT supplemented the independent efforts of the authors 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 a 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].