**PHENIKAA UNIVERSITY**

**FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING**



**FINAL REPORT**

**Deep Reinforcement Learning**

**Topic: Robot Learning to Walk and Balance Using Deep Reinforcement Learning (BipedalWalker)**

Student: Vũ Duy Minh Tiến

ID: 22014579

Class: K16 AIRB

Instructor: TS. Vũ Hoàng Diệu

*Ha Noi 2024*

**TABLE OF CONTENT**

Contents

**TABLE OF FIGURE**

# **1. INTRODUCTION**

**1.1. Problem**

Controlling a bipedal robot that can walk on two legs is one of the most complex problems in robotics and reinforcement learning. This task involves learning how to move forward efficiently while keeping the robot balanced, coordinating joint movements, and responding to uneven terrains or slopes. Traditional control algorithms struggle in this domain due to the need for explicit modeling and tuning for each situation. In contrast, deep reinforcement learning (DRL) offers a model-free approach that allows agents to learn from interaction with the environment.

In this project, the environment used is BipedalWalker-v3, an environment from OpenAI Gym that simulates a robot with four controllable joints. The agent must learn to control these joints to walk over challenging terrain without falling. This project aims to apply DRL algorithms to train an agent that can perform stable walking behavior in this environment.

**1.2. Motivation**

The motivation behind this project lies in the increasing interest in building autonomous robots capable of handling real-world tasks. Walking on two legs, like humans, requires a good sense of balance, adaptability to changes, and efficient movement planning. Solving this problem through DRL not only enhances our understanding of intelligent control systems but also brings us closer to building robust legged robots for practical applications in search and rescue, healthcare, or logistics.

Moreover, the BipedalWalker environment is an excellent benchmark for testing the effectiveness of various DRL algorithms due to its continuous action space and high-dimensional state space. Through this project, I hope to explore how different types of DRL methods perform and compare their advantages and limitations.

**1.3. My Idea**

My idea for this project is to compare two main approaches in reinforcement learning: value-based and policy-based methods. Specifically, I implement and train two agents:

* A Deep Q-Network (DQN) agent, which requires discretization of the continuous action space.
* A Proximal Policy Optimization (PPO) agent, which is designed for continuous action spaces and directly outputs a distribution over actions.

By running both agents on the same environment and comparing their performance in terms of total reward, learning speed, and stability, I hope to gain insights into which approach is more suitable for robotic walking tasks.

## **1.4. Workflow**

A diagram of a process

AI-generated content may be incorrect.

Figure 1:Workflow

This is the flowchart of the training process where a the optimizer:

The following diagram summarizes the general workflow of my project:

1. Initialize the BipedalWalker-v3 environment.
2. Initialize the agent (either DQN or PPO).
3. For each episode:
   * Reset the environment.
   * At each step:
     + Observe the current state.
     + Choose an action based on the current policy.
     + Apply the action to the environment.
     + Receive the next state and reward.
     + Store the transition (state, action, reward, next state).
   * Update the agent based on collected experiences.
4. Evaluate performance by plotting reward trends and comparing between methods.

# **2.** **ENVIRONMENT AND PREPROCESSING**

# **2.1. The BipedalWalker Environment**

A cartoon of a purple object on a green field

AI-generated content may be incorrect.

*Figure 2: Gym Bipedal Walker Enviroment*

BipedalWalker-v3 is a 2D physics simulation where a robot with two legs has to walk across a randomly generated terrain. Each leg has two joints, which are controlled by applying torques. The terrain includes flat surfaces, hills, and gaps that make walking more challenging. The agent receives observations about its current state and must decide what torques to apply to each of the four joints.

## **2.2. State and Action Space**

* State: A 24-dimensional continuous vector that includes the robot's position, velocity, angle, angular velocity, joint angles, joint speeds, and contact sensors with the ground.
* Action:
  + In PPO: A continuous 4-dimensional vector in the range [-1, 1], each representing the torque applied to a joint.
  + In DQN: Since DQN only supports discrete actions, the continuous space must be discretized. I created a fixed set of 16 action combinations.

## **2.3. Reward Function**

The reward is given to encourage forward movement and stable walking. The agent receives:

* Positive rewards for moving forward efficiently.
* Penalties for using too much torque or for falling down.
* A large bonus reward (+300) if the agent finishes the whole terrain without falling.

The reward design is important to ensure that the agent not only moves forward but also learns to maintain balance and walk economically

# **3. MODEL vs Algorithm**

## **3.1. Proximal Policy Optimization (PPO)**

PPO Agent Model Description

**3.1.1. Architecture**

The PPO agent is implemented using two neural networks: an actor and a critic.

The **actor network** receives the 24-dimensional state vector as input and outputs the mean and standard deviation of a Gaussian distribution over the 4 continuous actions.

The **critic network** estimates the state value V(s).

The typical architecture is:

Input layer: 24 units corresponding to the state vector.

Two hidden layers: Each with 64 or 128 units and ReLU activation.

Actor output: Two 4-dimensional vectors representing the mean and log-std of action distributions.

Critic output: Scalar value V(s).

**3.1.2. Training Algorithm**

The PPO training process involves the following:

Collect N trajectories (sequences of state, action, reward, next state, done).

Compute advantage estimates using GAE (Generalized Advantage Estimation).Perform multiple epochs of policy optimization by maximizing the clipped objective:

where r(θ) is the probability ratio between the new and old policies.

Update the critic by minimizing the squared error between predicted and target returns

* + 1. **Hyperparameters**
* Hidden units: 64–128
* Learning rate: 0.0003
* Discount factor γ: 0.99
* GAE lambda: 0.95
* Clip ratio ε: 0.2
* Epochs per update: 10
* Batch size: 64
  + 1. **Integration Points**
* State input: 24-dimensional vector from the environment.
* Action output: Mean and std for a Gaussian policy over 4 torque values.
* Training: Performed after collecting full episodes.
* Reward shaping: Encourages forward movement, energy efficiency, and penalizes instability.

**3.1.5. Benefits and Considerations**

Benefits:

* Works naturally with continuous action spaces.
* Stable learning via clipped updates.
* High sample efficiency and smooth convergence.

Considerations:

* Requires more memory to store full trajectories.
* More complex implementation due to dual networks and advantage estimation.

## **3.2. DQN**

# DQN Agent Model Description

## **1. Architecture**

The Deep Q-Network (DQN) agent is implemented as a multi-layer perceptron that approximates the action-value function . Given a state vector , the network outputs a Q-value for each action , where . The typical architecture is:

* **Input layer:** Dimension , corresponding to all state features (loss, accuracy, gradient norm, epoch index, etc.).
* **Hidden layers:** Two fully connected layers, each with units (e.g., ), followed by ReLU activations:
* **Output layer:** Fully connected layer with units, producing Q-values:

Here, are the network parameters.

## **3.2.1. Training Algorithm**

The DQN training follows the standard procedure:

1. **Experience Replay:** Store transitions in a replay buffer of fixed capacity. During learning, sample random minibatches to break temporal correlations.
2. **Target Network:** Maintain a separate target network with parameters . Every steps, update .
3. **Bellman Updates:** For each sampled transition, compute the target

* and minimize the mean squared error
* via gradient descent on .

1. **-Greedy Policy:** During training, choose actions according to:

* Decay from an initial value down to a minimum (e.g., 0.05) over time.

## **3.2.3. Hyperparameters**

* Hidden units : 64
* Learning rate : 0.001
* Discount factor : 0.99
* Replay buffer capacity: 5000 transitions
* Minibatch size: 32
* Target network update frequency : 20 updates
* Exploration decay:

## **3.2.4. Integration Points**

* **State input:** Concatenate ResNet-extracted feature statistics and training metrics into a vector of dimension .
* **Action output:** Two Q-values corresponding to optimizers SGD and SAM.
* **Training schedule:** After each epoch, compute reward and store transition, then perform one or more DQN updates.

## **3.2.5. Benefits and Considerations**

**Benefits:**

* Balances exploration of optimizer options with exploitation of learned policy.
* Uses off-policy learning (experience replay) for data efficiency.
* Stabilized by a fixed target network to prevent training divergence.

**Considerations:**

* Additional overhead from DQN updates on top of main training loop.
* Requires careful tuning of DQN hyperparameters to ensure convergence.

High-dimensional state vectors may lead to overfitting.

## **3.3. Algorithm Pseudocode**

PPO Training Loop:

* Collect trajectories for N episodes.
* Compute advantage estimates.
* Optimize policy and value networks using mini-batches.
* Clip updates to avoid large policy changes.

DQN Training Loop:

* For each episode, store transitions (s, a, r, s').
* Sample minibatches from the replay buffer.
* Compute target Q-values using Bellman equation.
* Update the Q-network using mean squared error.

**4. EVALUATION**

## **4.1. Training DQN model**

A graph of a chart

AI-generated content may be incorrect.

Figure 3: Reward per episode DQN

**Stability**: The total reward is quite stable, fluctuating mildly around −120 after the first ~1000 episodes.

**Learning Speed**: The agent learns quickly in the first ~500 episodes, after which the rewards stabilize and show little further improvement.

**Issue**: Some sharp reward drops (spikes) appear occasionally, possibly caused by temporary policy degradation during updates.

**Remark**: This model appears to have **converged early**, getting stuck in a local optimum without reaching rewards higher than −100.

## **4.2. PPO**

A graph showing a number of points

AI-generated content may be incorrect.

*Figure 4: Reward per episode PPO*

**Clear Progress**: Total reward steadily increases and surpasses 0, reaching around ~250 in the later episodes.

**High Variance**: The reward curve is highly jagged, showing strong fluctuations across training.

**Better Performance**: This PPO model **outperforms** the one in Chart 1 in terms of final reward, although it still requires further stabilization.

**Remark**: This is the **most effective PPO model** among the three, demonstrating good learning capabilities despite instability.

A graph of a graph showing a number of points

AI-generated content may be incorrect.

*Figure 5: Local optimum*

**Highly Unstable**: The rewards vary wildly between −300 and nearly 300, with no clear pattern.

**Slow Learning**: Reward only starts improving after ~800 episodes, indicating slower learning than PPO.

**No Clear Convergence**: Although some high rewards are achieved, the agent fails to maintain consistent performance.

**Remark**: DQN proves to be unstable, prone to overfitting, and struggles to converge in continuous environments like BipedalWalker due to its need to discretize the action space.

**5. COMPARISON**

**5.1 Convergence Speed**

PPO converges faster and reaches high performance around episode 500. DQN converges slowly and plateaus early.

**5.2 Final Performance**

PPO achieves high final rewards and stable walking. DQN sometimes learns basic walking but fails to generalize on harder terrain.

**5.3 Stability and Robustness**

PPO is more stable and shows less variation across different runs. DQN's performance varies depending on initialization and discretization.

# **6. CONCLUSION**

**6.1 Key Takeaways**

* Deep Reinforcement Learning enables agents to learn bipedal locomotion through direct interaction with the environment.
* Proximal Policy Optimization (PPO) clearly outperforms Deep Q-Network (DQN) in continuous control tasks such as BipedalWalker.
* PPO provides faster convergence, smoother policy updates, and more stable training behavior.
* Discretizing actions for DQN leads to instability and lower sample efficiency in continuous domains.

**6.2 Future Directions**

* Investigate advanced continuous control algorithms such as Soft Actor-Critic (SAC) or Twin Delayed DDPG (TD3).
* Explore curriculum learning strategies to gradually increase terrain difficulty and encourage progressive skill acquisition.
* Attempt sim-to-real transfer by testing trained agents on real-world robotic hardware.
* Experiment with hybrid actor-critic methods that combine value-based and policy-based learning for improved stability and performance.

# **REFERENCE**

1. OpenAI Gym Documentation
2. Schulman et al., "Proximal Policy Optimization Algorithms"
3. Mnih et al., "Playing Atari with Deep Reinforcement Learning"
4. PyTorch Official Tutorials