Implementing a DQN Agent for the CartPole-v1 Environment

1 Introduction

This report documents the implementation and evaluation of a Deep Q-Network (DQN) agent to solve the CartPole-v1 environment. The CartPole problem involves balancing a pole on a moving cart by applying forces to the cart. The goal is to keep the pole upright for as long as possible. The project began with the implementation of simple policies to understand the environment's behavior, followed by the development and training of a DQN agent to solve the environment.

2 Project Overview

2.1 Objectives

- Implement and evaluate simple policies for the CartPole-v1 environment.
- Develop a DQN agent to solve the environment.
- Train and evaluate the DQN agent's performance.
- Visualize the trained agent's interactions with the environment.

2.2 Tools and Libraries

- OpenAI Gym: For the CartPole-v1 environment.
- NumPy: For numerical computations.
- TensorFlow/Keras: For building and training the DQN model.
- Stable-Baselines3: For implementing the DQN agent.
- Matplotlib: For visualization.

3 Simple Policies Implementation

3.1 Policies

Four simple policies were implemented to understand the environment's behavior:

- **Angle-Based Policy:** Move left if the pole's angle is negative; otherwise, move right.
- **Position-Based Policy:** Move left if the cart's position is negative; otherwise, move right.
- **Velocity-Based Policy:** Move left if the cart's velocity is negative; otherwise, move right.
- **Combined Policy (Angle + Velocity):** Move left if both the pole's angle and cart's velocity are negative; move right if both are positive; otherwise, choose a random action.

3.2 Evaluation of Policies

Each policy was evaluated over 100 episodes, and the average reward was computed. The results are as follows:

Policy	**Average Reward**
Angle-Based Policy	42.42
Position-Based Policy	9.34
Velocity-Based Policy	9.45
Combined Policy	15.13

Table 1: Policy Evaluation Results

The Combined Policy performed the best, as it considered both the pole's angle and the cart's velocity, leading to more stable behavior.

4 DQN Agent Implementation

4.1 Model Architecture

The DQN agent was built using a neural network with the following architecture:

- **Input Layer:** 4 units (state space of CartPole-v1).
- **Hidden Layers:** 2 layers with 24 units each and ReLU activation.
- **Output Layer:** 2 units (action space of CartPole-v1) with linear activation.

4.2 Agent Configuration

- **Policy:** BoltzmannQPolicy for action selection based on Q-values.
- **Memory:** SequentialMemory with a limit of 50,000 steps.
- **Optimizer:** Adam optimizer with a learning rate of 0.001.

4.3 Training

The agent was trained for 50,000 timesteps. The training process involved:

- Collecting experiences (state, action, reward, next state) and storing them in memory.
- Sampling mini-batches from memory to update the Q-network.
- Using the target network to stabilize training.

5 Visualization

The trained agent's performance was visualized by running it for 20 episodes. The agent demonstrated stable and efficient balancing of the pole, consistently achieving high rewards.

6 Challenges and Learnings

6.1 Challenges

- **Reward Sparsity:** The environment provides a reward of +1 for every timestep the pole remains upright, making it challenging to learn long-term strategies.
- **Hyperparameter Tuning:** Selecting appropriate hyperparameters (e.g., learning rate, memory size) was crucial for the agent's performance.

6.2 Learnings

- The importance of experience replay and target networks in stabilizing DQN training.
- The effectiveness of combining multiple state features (e.g., angle and velocity) in improving policy performance.

7 Conclusion

This project successfully implemented and evaluated a DQN agent to solve the CartPole-v1 environment. The agent achieved near-optimal performance, demonstrating the effectiveness of deep reinforcement learning in solving control problems. Future work could explore:

- **Advanced Algorithms:** Implementing algorithms like Double DQN or Dueling DQN for improved performance.
- **Hyperparameter Optimization:** Using techniques like grid search or Bayesian optimization to fine-tune hyperparameters.
- **Transfer Learning:** Applying the trained agent to more complex environments.

8 References

- OpenAI Gym Documentation: https://www.gymlibrary.dev/
- Stable-Baselines3 Documentation: https://stable-baselines3.readthedocs.io/
- Mnih, V., et al. (2015). *Human-level control through deep reinforcement learning.* Nature.

Humanoid Walking Simulation Using Reinforcement Learning

1 Introduction

This report documents the development and implementation of a 2D humanoid walking simulation using reinforcement learning (RL). The goal of the project was to train a humanoid agent to walk efficiently in a simulated environment by leveraging RL techniques. The simulation was built using the Box2D physics engine and Pygame for rendering, while the RL model was implemented using Stable-Baselines3 with the PPO (Proximal Policy Optimization) algorithm.

The project involved designing a custom environment, implementing a reward function, training the RL model, and evaluating its performance. This report covers the key aspects of the project, including the simulation setup, RL implementation, reward function design, and results.

2 Project Overview

2.1 Objectives

- Develop a 2D humanoid walking simulation.
- Implement a reinforcement learning model to control the humanoid's walking motion.
- Design a reward function to encourage stable and efficient walking.
- Train and evaluate the model's performance.

2.2 Tools and Libraries

- Box2D: Physics engine for simulating the humanoid's movements.
- Pygame: Rendering engine for visualizing the simulation.
- Stable-Baselines3: RL library for implementing the PPO algorithm.
- Gymnasium: Framework for creating custom RL environments.
- NumPy: For numerical computations.

3 Simulation Setup

3.1 Humanoid Model

The humanoid model consists of the following components:

- **Torso**: The main body of the humanoid.
- **Thighs and Shins**: Leg segments connected by revolute joints.
- **Joints**: Four joints (two hips and two knees) controlled by motors.

The humanoid was modeled using Box2D, with each body part represented as a dynamic body. The joints were configured to allow rotational movement, enabling the humanoid to walk.

3.2 Environment

The simulation environment was designed as a 2D plane with the following features:

- **Ground**: A static body representing the floor.
- **Flag**: A visual marker indicating the goal position.
- **Humanoid**: The agent controlled by the RL model.

The environment was rendered using Pygame, with a scaling factor of 100 pixels per meter (PPM) to convert Box2D coordinates to screen coordinates.

4 Reinforcement Learning Implementation

4.1 Custom Gym Environment

A custom Gymnasium environment (`HumanoidEnv`) was created to interface with the simulation. The environment defines the following:

- **Action Space**: A continuous space representing motor speeds for the four joints.
- **Observation Space**: A continuous space representing the humanoid's state, including joint angles, positions, and velocities.
- **Reward Function**: A custom function to compute rewards based on the humanoid's performance.
- **Termination Condition**: The episode terminates if the humanoid falls or reaches the goal.

4.2 PPO Algorithm

The PPO (Proximal Policy Optimization) algorithm was chosen for training due to its stability and efficiency in handling continuous action spaces. Key hyperparameters included:

- Learning Rate: 0.3
- Gamma (Discount Factor): 0.99
- Number of Steps per Rollout: 2048
- Batch Size: 64
- Number of Epochs per Update: 10

4.3 Training Process

The model was trained for 50,000 timesteps using the following steps:

1. Initialize the environment and model.
2. Collect observations and compute actions using the current policy.
3. Step the environment and compute rewards.
4. Update the policy using PPO.
5. Repeat until the episode terminates or the timestep limit is reached.

5 Reward Function Design

The reward function was designed to encourage stable and efficient walking. It consists of the following components:

5.1 Forward Progress

- **Reward**: Positive reward for moving forward.
- **Penalty**: Negative reward for moving backward.

5.2 Balance Maintenance

- **Reward**: Positive reward for maintaining the torso height above a threshold.

5.3 Smooth Movement

- **Penalty**: Negative reward for large joint velocities to encourage smooth motion.

5.4 Gait Stability

- **Reward**: Positive reward for alternating leg movements to encourage a natural gait.

5.5 Falling Penalty

- **Penalty**: Large negative reward if the humanoid falls.

5.6 Combined Reward

The total reward is a weighted sum of the above components:

$$\text{Reward} = \text{Forward Progress} + \text{Balance Reward} + \text{Gait Reward} + \text{Smoothness Penalty} + \text{Falling Penalty} \tag{1}$$

6 Results and Evaluation

6.1 Training Performance

- The model was able to learn a stable walking gait after approximately 20,000 timesteps.
- The humanoid achieved forward progress while maintaining balance and avoiding falls.

6.2 Testing Performance

- The trained model was tested in the simulation environment.
- The humanoid successfully walked to the goal position without falling in most episodes.

7 Conclusion

This project demonstrated the application of reinforcement learning to control a 2D humanoid walking simulation. Future work could explore:

- 3D Simulation: Extending the simulation to three dimensions.
- Advanced RL Algorithms: Experimenting with SAC or DDPG.
- Improved Reward Function: Incorporating energy efficiency metrics.

8 References

- Stable-Baselines3 Documentation: https://stable-baselines3.readthedocs.io/
- OpenAI Spinning Up in Deep RL: https://spinningup.openai.com/
- Box2D Documentation: https://box2d.org/documentation/