Training a RL Agent in BipedalWalker-v3 with TD3 approach

Mohammad Rouintan

Computer Science, Shahid Beheshti University

Abstract

This report details the implementation of a reinforcement learning (RL) agent designed to master the BipedalWalker-v3 environment, a complex and demanding task within the OpenAI Gym suite. Utilizing the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm, the agent aims to control a bipedal robot to navigate uneven terrain efficiently. The project involves setting up the environment, implementing the TD3 algorithm using PyTorch, training the agent with techniques such as reward shaping and experience replay, and evaluating its performance. The results highlight the agent's ability to learn effective walking strategies, demonstrating the potential of advanced RL algorithms in solving intricate control problems. This report outlines the methodology, challenges, solutions, and performance analysis of the RL agent, offering insights into the application of TD3 in continuous action spaces.

Introduction

Reinforcement learning (RL) has emerged as a powerful approach in the field of machine learning, particularly for solving complex control tasks. The BipedalWalker-v3 environment, part of the OpenAI Gym suite, presents a formidable challenge by requiring an agent to control a bipedal robot traversing uneven terrain. This task not only tests the agent's ability to learn and adapt but also serves as a benchmark for advanced RL algorithms.

In this assignment, we focus on developing an RL agent capable of mastering the BipedalWalker-v3 environment using the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm. TD3 is a state-of-the-art RL algorithm designed for continuous action spaces, addressing the limitations of its predecessor, Deep Deterministic Policy Gradient (DDPG), by introducing mechanisms to reduce function approximation error and improve learning stability.

The project is structured as follows:

1. Environment Setup: Installation of necessary dependencies and a thorough understanding of the BipedalWalker-v3 environment, including its observation and action spaces.
2. Algorithm Implementation: Selection and implementation of the TD3 algorithm using PyTorch, encompassing policy and value networks, and appropriate loss functions.
3. Agent Training: Setting up the training process with techniques such as reward shaping, curriculum learning, experience replay, and action noise to enhance exploration and prevent suboptimal convergence.
4. Evaluation: Conducting multiple evaluation episodes, visualizing training progress, and recording the agent's performance to assess its ability to navigate the terrain.

This report aims to provide a detailed account of the methodologies employed, the challenges encountered, and the solutions implemented to develop a robust RL agent. By analyzing the agent's performance and identifying potential areas for improvement, we seek to contribute valuable insights into the application of advanced RL algorithms in complex environments.

Methodologies

About BipedalWalker-v3 Environment

The BipedalWalker-v3 environment is a highly challenging and intricate environment within the OpenAI Gym suite designed to test the capabilities of reinforcement learning (RL) agents. This environment simulates a bipedal robot that must learn to walk across uneven terrain, requiring sophisticated control strategies and the ability to adapt to varying conditions. The primary objective for the RL agent is to enable the robot to move forward as efficiently and stably as possible, navigating through obstacles and maintaining balance on rugged surfaces.

Observation Space:

The observation space of the BipedalWalker-v3 environment is a 24-dimensional continuous vector that provides detailed information about the state of the bipedal robot and its interaction with the environment. The components of this observation space include:

- Hull Angle and Angular Velocity: The orientation and rotational speed of the robot's main

body, which are crucial for maintaining balance.
- Joint Angles and Angular Velocities: The angles and rotational speeds of the joints in the robot's legs, providing information on the configuration and movement of the limbs.
- Leg Contact Points: Indicators of whether each leg is in contact with the ground, essential for determining support and stability.
- Velocities: Linear velocities of various parts of the robot, indicating the speed and direction of movement.
- Lidar Measurements: Distances to the ground from several points around the robot's hull, simulating a Lidar sensor that helps in perceiving the terrain ahead.

These observations collectively give the agent a comprehensive view of the robot's current state and its interaction with the environment, enabling it to make informed decisions about the actions to take.

Action Space:

The action space of the BipedalWalker-v3 environment consists of a 4-dimensional continuous vector. Each element of this vector represents a control input to one of the robot's four motorized joints. The components are:

- Hip Joint (Left Leg): Torque applied to the hip joint of the left leg.
- Knee Joint (Left Leg): Torque applied to the knee joint of the left leg.
- Hip Joint (Right Leg): Torque applied to the hip joint of the right leg.
- Knee Joint (Right Leg): Torque applied to the knee joint of the right leg.

Each action value is typically bounded within a range, for example, between -1 and 1, where the sign and magnitude of the value indicate the direction and intensity of the torque applied. The agent must learn to coordinate these torques to achieve smooth and stable walking, which involves complex control strategies to handle the dynamics of the bipedal robot.

About Twin Delayed Deep Deterministic Policy Gradient (TD3)

The Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm is an enhancement of the Deep Deterministic Policy Gradient (DDPG) algorithm, specifically designed to improve the stability and performance of training in continuous action spaces. TD3 addresses several key issues inherent in DDPG, such as overestimation bias and the instability of function approximation. Here, we delve into the details of the TD3 algorithm, highlighting its unique features and the mathematical foundations that underpin its operation.

Key Features of TD3:

- Clipped Double Q-Learning: TD3 uses two Q-networks instead of one to mitigate overestimation bias. By taking the minimum value between the two Q-networks, TD3 ensures a more conservative estimate of the value function.
- Delayed Policy Updates: The policy network is updated less frequently than the Q-networks to reduce variance and improve stability.
- Target Policy Smoothing: TD3 adds noise to the target policy during training to smooth the value estimation, which helps in reducing the variance of the target values and prevents the exploitation of sharp peaks in the Q-function.

Algorithm Overview:

1. Initialization
   - Initialize two Q-networks $Q_{\theta_1}(s,a)$ and $Q_{\theta_2}(s,a)$ with random parameters θ_1 and θ_2.
   - Initialize the policy network $\pi_\phi(s)$ with random parameters ϕ.
   - Initialize target networks $Q_{\theta'_1}(s,a)$, $Q_{\theta'_2}(s,a)$, and $\pi_{\phi'}(s)$ with $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$.
   - Initialize an empty replay buffer $\mathcal{D}$.
2. Interaction with the Environment
   For each time step t:
   - Select action $a_t = \pi_\phi(s_t) + \epsilon$, where $\epsilon \sim \mathcal{N}(0,\sigma)$ (exploration noise).
   - Execute action a_t and observe reward r_t and next state s_{t+1}.
   - Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$.
3. Learning from Experience
   For each gradient step:
   - Sample a random mini-batch of N transitions (s_i, a_i, r_i, s_{i+1}) from the replay buffer $\mathcal{D}$.
   - Update Q-Networks:
     - Compute target actions with added noise: $a'_i = \pi_{\phi'}(s_{i+1}) + \epsilon$, where $\epsilon \sim \text{clip}(\mathcal{N}(0,\sigma_{\text{target}}), -c, c)$.
     - Compute target Q-values:

       $$y_i = r_i + \gamma \min(Q_{\theta'_1}(s_{i+1}, a'_i), Q_{\theta'_2}(s_{i+1}, a'_i))$$

     - Update Q-networks by minimizing the loss:

       $$L(\theta_j) = \frac{1}{N}\sum_i (Q_{\theta_j}(s_i,a_i) - y_i)^2 \quad \text{for} \quad j = 1,2$$

- Delayed Policy Update:
  If $t \mod d = 0$ (where d is the delay parameter):
  - Update policy network by maximizing the expected Q-value:

$$\nabla_\phi J(\phi) = \frac{1}{N} \sum_i \nabla_a Q_{\theta_1}(s_i, a)|_{a=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i)$$

  - Update target networks with soft updates:

$$\theta'_j \leftarrow \tau\theta_j + (1-\tau)\theta'_j \quad \text{for} \quad j = 1, 2$$

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

The TD3 algorithm enhances the stability and performance of the DDPG algorithm through several innovative techniques. By using twin Q-networks, delaying policy updates, and smoothing the target policy, TD3 effectively mitigates common issues in training RL agents in continuous action spaces. This makes TD3 a robust choice for complex environments like BipedalWalker-v3, where precise and stable control is paramount.

Components of code and model architecture

Model Class:

The 'Model' class represents a neural network used for either the policy (actor) or value (critic) networks in the TD3 algorithm. It inherits from 'nn.Module', a base class for all neural network modules in PyTorch.

Key Components:

- Initialization ('__init__'):*Initializes the network with specified neurons, activation function for the final layer, learning rate, and device (CPU or GPU).
- Layer Initialization ('_init_layers'): Constructs the layers of the network based on the provided neuron configuration. Each layer is a linear transformation followed by a ReLU activation function, except for the final layer, which uses the specified activation function.
- Optimizer Initialization ('_init_optimizer'): Sets up the optimizer (Adam) for training the network.
- Forward Pass ('forward'):*Defines the forward pass of the network, which computes the output given an input 'x'.

Memory Class:

The 'Memory' class acts as a replay buffer for storing past experiences (transitions) during training. This buffer is essential for the stability of training as it allows the agent to learn from a diverse set of past experiences.

Key Components:

- Initialization ('__init__'): Initializes the replay buffer with the given size, device, and dimensions for observations and actions.
- Learn Method ('learn'): Stores a new transition (state, action, reward, next state, and done flag) in the buffer.
- Get Batch Method ('get_batch'): Samples a random mini-batch of transitions from the buffer for training the networks.

Agent Class:

The 'Agent' class encapsulates the TD3 algorithm, managing the interaction with the environment, learning process, and model updates.

Key Components:

- Initialization ('__init__'):** Sets up the environment, hyperparameters, device, and loads models if available.
- Get Noisy Action ('get_noisy_act'): Returns a noisy action for exploration by adding Gaussian noise to the deterministic action from the actor network.
- Get Action ('get_act'): Returns the deterministic action from the actor network.
- Q-Loss Calculation ('q_loss'): Computes the loss for the Q-networks.
- Policy Loss Calculation ('policy_loss'): Computes the loss for the policy network.
- Find Targets ('find_targets'): Computes target Q-values using the target policy and value networks with added noise for target policy smoothing.
- Update ('update'): Updates the Q-networks and, optionally, the policy network based on sampled mini-batch from memory.
- Update Final Model ('update_final_model'):** Soft updates the target networks.
- Load Models ('load_models'): Loads models and memory from files if they exist.
- Save Models ('save_models'): Saves models and memory to files.

The provided code comprises three primary classes crucial for implementing the TD3 algorithm. The 'Model' class defines the neural network structure for the actor and critic networks. The 'Memory' class implements a replay buffer for storing and sampling transitions. Finally, the 'Agent' class orchestrates the interaction with the environment, manages the learning process, and updates the models based on sampled experiences. Together, these classes enable the implementation of a robust reinforcement learn-

ing agent capable of learning in continuous action spaces.

Agent Training

The provided code implements the training loop for a TD3 (Twin Delayed Deep Deterministic Policy Gradient) agent in the 'BipedalWalker-v3' environment. Below is a detailed explanation of each part of the code:

Configuration and Initialization:

- Hyperparameters:
  - 'batch_size': Number of samples used in each training step.
  - 'lr': Learning rate for the optimizer.
  - 'tau': Soft update parameter for the target networks.
  - 'act_sigma': Standard deviation of the noise added to the policy actions for exploration.
  - 'tr_sigma': Standard deviation of the noise added to the target actions for policy smoothing.
  - 'tr_clip': Clipping range for the target action noise.
  - 'gamma': Discount factor for future rewards.
  - 'policy_delay': Delay in policy updates to stabilize training.
  - 'total_episode': Total number of episodes for training.
  - 'render': Boolean to control the rendering of the environment.
- Environment Creation:
  - 'create_env(name)': Function to create and initialize the environment with the specified name.
  - 'env = create_env('BipedalWalker-v3')': Create the 'BipedalWalker-v3' environment.
- Agent Initialization:
  - 'agent = Agent(env, lr, gamma, tau)': Initialize the TD3 agent with the environment and specified hyperparameters.

Training Loop:

The main training loop consists of the following steps:

- Episode Initialization:
  - Start a new episode if the agent has not reached the total number of episodes.
  - 'action = agent.get_noisy_act(state, act_sigma)': Get a noisy action from the agent's policy for exploration.
  - 'nextState, reward, terminated, truncated, info = env.step(action)': Take a step in the environment using the chosen action.
- Memory Update:
  - 'agent.memory.learn(state, action, reward, nextState, terminated)': Store the transition in the replay buffer.
- Episode Termination Handling:
  If the episode is terminated or truncated, perform the following:
  - Record the time elapsed.
  - Increment the episode counter.
  - Reset the environment and initialize the state.
  - Test the agent's performance in the environment without noise.
  - Update the average reward using exponential smoothing.
  - Log the episode data to the CSV file.
  - Save the agent's models.
  - Print the episode summary.
- State Update: If the episode is not terminated, update the current state to the next state.
- Policy and Q-Network Update:
  - Determine if the policy should be updated based on the 'policy_delay'.
  - 'agent.update(batch_size, tr_sigma, tr_clip, should_update_policy)': Update the agent's networks using a batch of transitions from the replay buffer.

During training, the agent takes actions with added noise ('act_sigma') to encourage exploration. Transitions are stored in the replay buffer, and the agent's networks are updated using sampled transitions. At the end of each episode, the agent's performance is evaluated without noise to get an accurate measure of its capabilities. The cumulative reward for the test run is calculated and used to update the average reward.

This code sets up and runs a TD3 agent in the 'BipedalWalker-v3' environment, managing both the training and evaluation phases, and ensuring the agent's progress is saved and logged for future reference.

Results

Learning curve plots

In this section, we have shown learning curve plots for model.

Based on the learning curve , we can see that the model is still capable of training, but due to the

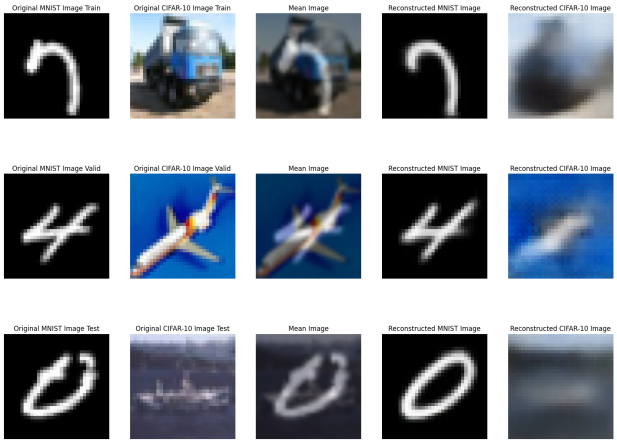

Figure 2: *Result of approach 1*

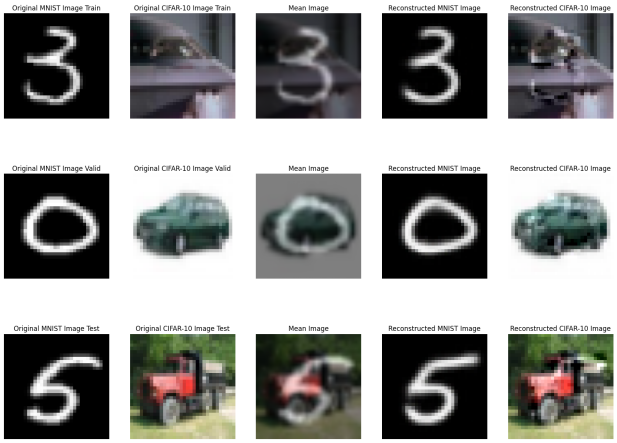

Figure 3: *Result of approach 2*

Figure 1: *Animated GIF included in LaTeX*

time-consuming training process, only 50 epochs of training have been completed.

Final results

Our model has one encoder and two decoders, the encoder takes the mean image as input, and one decoder outputs the MNIST image and the other one outputs the CIFAR10 image. After going through the training process in this project, we will compare the model based on two different approaches. In the first approach, the model is checked based on the output of both decoders, that is, the output of both decoders is checked based on criteria such as SSIM and PSNR. But because the trained model is strong in reconstructing MNIST images and weak in reconstructing CIFAR10 images, we first reconstruct the MNIST image by decoder related to reconstructing MNIST images. Then, according to the following formula, we reconstruct the CIFAR10 image by the mean image and the reconstructed MNIST image:

$$cifar\_image = 2 \times mean\_image - mnist\_image$$

Table 1: *Results of approach 1*

Phase	MSELoss	SSIM	PSNR
Train	0.02	0.65	19.63
Validation	0.02	0.65	19.61
Test	0.03	0.62	19.0

Table 2: *Results of approach 2*

Phase	MSELoss	SSIM	PSNR
Train	0.01	0.84	21.83
Validation	0.01	0.83	21.75
Test	0.02	0.82	21.17

According to Figure 5, it is clear that the decoder related to the reconstruction of MNIST images has worked very well, but the decoder related to the reconstruction of CIFAR10 images displays the images smoothly. But according to figure 6 both reconstructed images in Approach 2 are very similar to the original images.

Conclusion

The aim of this project is to design an auto-encoder to reconstruct MNIST images and CIFAR10 from their mean image.

In this project, we have successfully demonstrated the application of autoencoders in reconstructing images from two distinct datasets—MNIST and CIFAR-10—using their element-wise mean. The autoencoder model was meticulously designed, integrating a pre-trained ResNet-50 for the encoder and custom-built convolutional and deconvolutional layers for the decoders. This architecture allowed us to harness the robust feature extraction capabilities of ResNet-50 while enabling detailed reconstruction through the decoders.

The preprocessing steps, including resizing and

channel extension of MNIST images , were crucial in ensuring the compatibility of inputs and the stability of the training process. By carefully preparing the datasets and employing appropriate transformations, we created a uniform input pipeline that facilitated effective training of the autoencoder.

The results of the model were evaluated both visually and quantitatively. The reconstructions of the MNIST and CIFAR-10 images from their mean images showcased the potential of autoencoders in cross-domain image synthesis. The dual-decoder approach proved to be effective in handling the distinct characteristics of the two datasets, providing specialized pathways for each image type.

But in the end, better results have been obtained by using the second approach