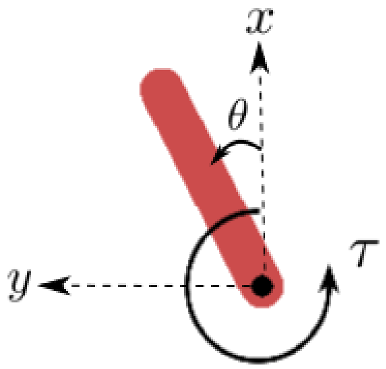
INFO8003-1: Optimal Decision Making for Complex Problems



Assignment 3: Searching High-Quality Policies to Control an Unstable Physical System

Staff :
ERNST Damien, *Teacher*
LOUETTE Arthur, *Teaching Assistant*
MIFTARI Bardhyl, *Teaching Assistant*

Group :
LAMBERMONT Romain
LOUIS Arthur

May 17, 2024

Contents

1 Description of the Environments

For this assignment, we consider two different environments from the `Gymnasium` package: `InvertedPendulum-v4` and `InvertedDoublePendulum-v4`. Our goal is to control these environments to maximize our rewards by keeping them in acceptable states around equilibrium. The explanations gathered about these environments can be found on the official website of the Farama Foundation. In this section, we will briefly explain how these domains work.

1.1 `InvertedPendulum-v4`

- **Action Space:**
  - The action space is continuous.
  - The agent takes a 1-element vector representing the force applied to the cart.
  - The force can range from -3 to 3, where the magnitude represents the amount of force and the sign represents the direction.
- **Observation Space:**
  - The state space consists of positional values of the pendulum system's body parts, followed by their velocities.
  - The observation is a ndarray with shape (4,).
  - **Details:**
    - Position of the cart along the linear surface
    - Vertical angle of the pole on the cart
    - Linear velocity of the cart
    - Angular velocity of the pole on the cart
- **Rewards:**
  - The goal is to keep the inverted pendulum upright within a certain angle limit.
  - A reward of $+1$ is awarded for each timestep the pole remains upright.
- **Starting State:**
  - All observations start in the state $(0.0, 0.0, 0.0, 0.0)$.
  - Uniform noise in the range of [-0.01, 0.01] is added to the values for stochasticity.
- **Episode End:**
  - The episode ends when any of the following conditions are met:
    - **Truncation:** The episode duration reaches 1000 timesteps.
    - **Termination:** Any of the state space values is no longer finite.
    - **Termination:** The absolute value of the vertical angle between the pole and the cart is greater than 0.2 radians.

1.2 `InvertedDoublePendulum-v4`

- **Action Space:**
  - The action space is continuous.
  - The agent takes a 1-element vector representing the force applied to the cart.
  - The force can range from -1 to 1, where the magnitude represents the amount of force and the sign represents the direction.

- **Observation Space:**
  - The state space consists of positional values of the pendulum system's body parts, followed by their velocities.
  - The observation is a ndarray with shape (11,).
  - **Details:**
    * Position of the cart along the linear surface
    * Sine of the angle between the cart and the first pole
    * Sine of the angle between the two poles
    * Cosine of the angle between the cart and the first pole
    * Cosine of the angle between the two poles
    * Velocity of the cart
    * Angular velocity of the angle between the cart and the first pole
    * Angular velocity of the angle between the two poles
    * Constraint force - 1
    * Constraint force - 2
    * Constraint force - 3
- **Rewards:**
  - The reward consists of three parts:
    * **Alive bonus:** A reward of $+10$ is awarded for each timestep the second pole remains upright.
    * **Distance penalty:** Calculated as $0.01 \cdot x^2 + (y-2)^2$, where x and y are the coordinates of the tip of the second pole.
    * **Velocity penalty:** A negative reward penalizing fast movements, calculated as $0.001 \cdot v_1^2 + 0.005 \cdot v_2^2$.
  - The total reward is given by:

$$\text{reward} = \text{alive\_bonus} - \text{distance\_penalty} - \text{velocity\_penalty}$$

- **Starting State:**
  - All observations start in the state $(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)$.
  - Uniform noise in the range of [-0.1, 0.1] is added to the positional values (cart position and pole angles).
  - Standard normal noise with a standard deviation of 0.1 is added to the velocity values.
- **Episode End:**
  - The episode ends when any of the following conditions are met:
    * **Truncation:** The episode duration reaches 1000 timesteps.
    * **Termination:** Any of the state space values is no longer finite.
    * **Termination:** The y-coordinate of the tip of the second pole is less than or equal to 1.

2 Fitted Q Iteration (FQI)

Fitted Q Iteration (FQI) is an off-policy batch reinforcement learning algorithm designed to effectively manage large state and action spaces. Unlike online learning algorithms that update value estimates based on individual transitions, FQI utilizes batches of transitions to iteratively update the Q-function, making it particularly suitable for complex environments where acquiring new data is costly.

2.1 Implementation Overview

The FQI implementation employs a neural network to approximate the Q-function. This network inputs state and action pairs and outputs estimated Q-values. The algorithm progresses through multiple epochs, training the network to minimize the difference between the current Q-values and the target Q-values derived from the Bellman optimality equation.

2.1.1 Network Architecture

The network, implemented using PyTorch, consists of four fully connected layers with Tanh activation functions to ensure bounded outputs, critical in environments with continuous action spaces. The architecture is defined as follows:

- First layer: Maps the input state-action pair to a 20-dimensional hidden space.
- Second and third layers: Each layer maps the 20-dimensional space to another 20-dimensional space, facilitating deeper feature extraction.
- Output layer: Maps the final hidden state to a single output, representing the Q-value for the input state-action pair.

2.1.2 Training Process

The training process involves generating a dataset of one-step system transitions from the environment. Each transition includes the current state, the action taken, the reward received, the next state, and whether the episode has ended. Training the Q-network with this data approximates the optimal Q-function by:

1. Preparing batches of state-action pairs.
2. For each batch, computing the target Q-values using the reward and the maximum Q-value of the next state, adjusted by the discount factor.
3. Updating the Q-network by minimizing the mean squared error between the network's output and the target Q-values.

2.1.3 Operational Flow

- Initialization: Set up the environment and the Q-network. Preload and preprocess the transition samples.
- Loop through epochs:
  - For each batch of data, perform a forward pass through the network to get the current Q-values.
  - Compute the target Q-values and perform a backward pass to update the network weights.
- After training, the model is evaluated on the environment to verify its performance, typically measured by the cumulative reward obtained from deploying the trained policy.

2.2 Execution Details

This implementation is designed to be flexible, with the capability to train on different environments by simply adjusting the configuration parameters such as the learning rate, batch size, and number of epochs. The use of PyTorch and its efficient computation capabilities on GPUs ensures that the training is performed swiftly, even with large batches and complex network architectures.

2.3 Observations from Training Plots

The training results from the FQI application on two environments, InvertedPendulum-v4 and InvertedDoublePendulum-v4, provide valuable insights:

- **InvertedPendulum-v4**: The cumulative rewards show a peak performance at certain transition sample sizes, indicating a potential optimal batch size for training convergence. Notably, a significant variability in performance suggests sensitivity to initial conditions or stochastic elements within the training process. The cumulative reward during training can be seen on figure 1.
- **InvertedDoublePendulum-v4**: This environment displays more fluctuating performance, with multiple peaks indicating the complex dynamics involved. The overall lower cumulative rewards compared to the InvertedPendulum-v4 might reflect the increased difficulty of stabilizing a double pendulum system. The cumulative reward during training can be seen on figure 2.

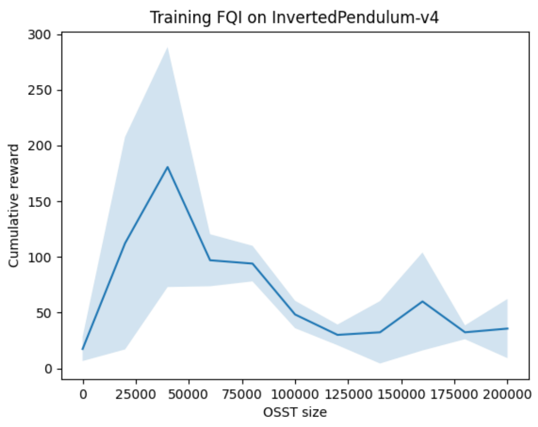


Figure 1: Evolution of the cumulative reward during training for the simple inverted pendulum

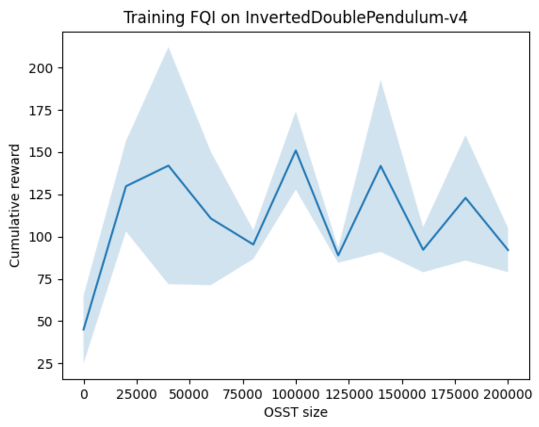


Figure 2: Evolution of the cumulative reward during training for the double inverted pendulum

3 REINFORCE Algorithm

REINFORCE is a policy gradient method used in reinforcement learning that directly optimizes the policy by updating it in a direction that maximizes the expected reward. It operates on an on-policy basis, using the generated episodes to update the policy that is being learned.

3.1 Implementation Overview

The REINFORCE implementation leverages a neural network to model the policy. The network takes the state as input and outputs a probability distribution over actions. The goal is to train the network to increase the probability of actions that lead to higher returns.

3.1.1 Network Architecture

The network, developed using PyTorch, includes several fully connected layers, each followed by non-linear activation functions to facilitate complex pattern learning in action decision processes. The specific architecture is structured as follows:

- First layer: Processes the state input into a 32-dimensional hidden layer.
- Intermediate layers: Each layer progressively refines the learned features, typically extending to two or three more layers depending on the complexity of the environment.

- Output layer: Outputs a probability distribution over possible actions using a softmax activation function, ensuring all action probabilities sum to one.

3.1.2 Training Process

Training in REINFORCE involves generating episodes using the current policy and using the complete episodes to update the policy based on the rewards obtained. The steps include:

1. Generate episodes by following the current policy.
2. For each action taken in an episode, calculate the return from that state till the end of the episode.
3. Update the policy by maximizing the log-probability of the actions taken, weighted by the calculated returns.

3.1.3 Operational Flow

- Initialization: Prepare the environment and initialize the policy network.
- Training loop:
  - Generate episodes using the policy network.
  - For each step in an episode, compute updates to the policy using the observed returns.
  - Apply gradient ascent to adjust the policy network parameters.
- Evaluate the performance of the updated policy by measuring the cumulative reward across new episodes.

3.2 Execution Details

The flexibility of the REINFORCE algorithm allows adaptation across various environments. Adjustments can be made to the learning rate and network architecture based on specific domain requirements. The algorithm's dependency on full episodes makes it particularly effective in episodic tasks.

3.3 Observations from Training Plots

The training results using the REINFORCE algorithm on the InvertedPendulum-v4 and InvertedDoublePendulum-v4 environments illustrate several key points:

- **InvertedPendulum-v4**: The rewards exhibit high variability with intermittent peaks, suggesting the algorithm's sensitivity to initial conditions and the stochastic nature of the policy exploration. This behavior is depicted in figure 3.
- **InvertedDoublePendulum-v4**: The performance is less stable with frequent fluctuations in rewards, reflecting the complexity and challenge of balancing a double pendulum. This is illustrated in figure 4.

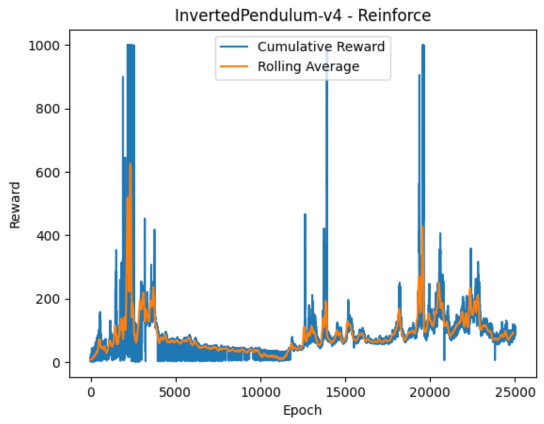


Figure 3: Training dynamics of REINFORCE on InvertedPendulum-v4

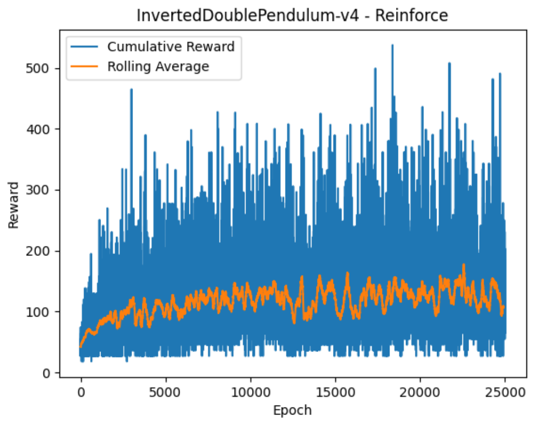


Figure 4: Training dynamics of REINFORCE on InvertedDoublePendulum-v4

4 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. It combines ideas from DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network) to learn policies that maximize the expected return.

4.1 Implementation Overview

DDPG employs two separate networks: an actor that specifies the current policy by mapping states to actions, and a critic that evaluates the action by computing the value function. Both networks are trained simultaneously to improve policy performance and value estimation.

4.1.1 Network Architecture

The architecture for both networks involves multiple layers with ReLU activations to foster non-linearity, ensuring effective function approximation:

- Actor Network:
  - Input layer: Takes the state as input.
  - Hidden layers: Multiple layers transforming the input into action values.
  - Output layer: Outputs a continuous action value using a tanh function to bound the actions.
- Critic Network:
  - Input layer: Accepts both state and action as input.
  - Hidden layers: Processes the combined information to estimate Q-values.
  - Output layer: Provides a single value output estimating the Q-value.

4.1.2 Training Process

DDPG updates the policy and value networks using sampled data from the environment, with the critic learning to evaluate the policy's action by minimizing the loss between predicted and target Q-values, and the actor updating the policy to maximize the Q-value predicted by the critic:

1. Collect transitions by following the current policy and store them in a replay buffer.

2. Sample a random minibatch of transitions from the replay buffer.
3. Update the critic by minimizing the mean squared loss between the predicted Q-values and the target Q-values computed using the Bellman equation.
4. Update the actor using the sampled policy gradient.

4.1.3 Operational Flow

- Initialization: Set up the environment, actor and critic networks.
- Training Loop:
  - Sample data from the environment according to the current policy.
  - Perform batch updates to the actor and critic networks based on the data sampled.
  - Occasionally sync the target networks with the main networks.
- Evaluate the policy by measuring the performance in terms of cumulative rewards after sufficient training epochs.

4.2 Execution Details

DDPG is particularly adaptable to any environment with continuous action spaces, requiring fine-tuning of hyperparameters like learning rate, update frequency, and the size of the replay buffer to optimize performance.

4.3 Observations from Training Plots

The results of applying DDPG to the InvertedPendulum-v4 and InvertedDoublePendulum-v4 environments provide significant insights:

- **InvertedPendulum-v4**: Shows fluctuations in mean cumulative rewards across epochs, with several peaks suggesting moments of optimal policy performance before potential divergence or exploration variance. See Figure 5.
- **InvertedDoublePendulum-v4**: Demonstrates higher peaks and more pronounced variability, indicating the complex nature of achieving stability in more challenging environments. See Figure 6.

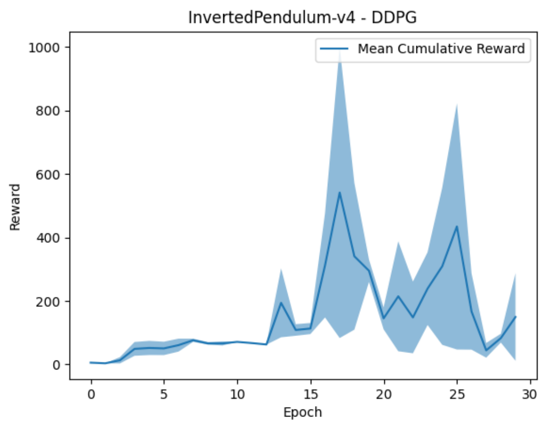


Figure 5: Performance of DDPG on InvertedPendulum-v4

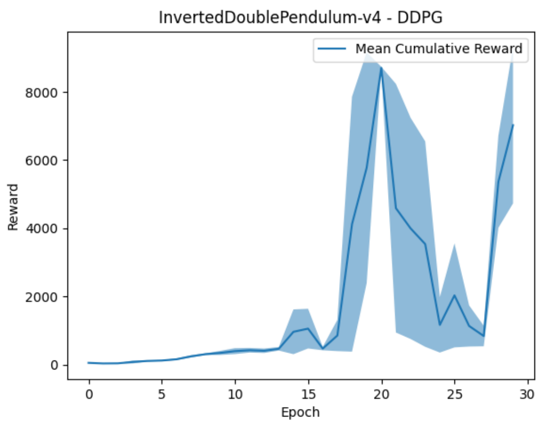


Figure 6: Performance of DDPG on InvertedDoublePendulum-v4

5 Comparison with `Stable-Baselines3`