CSE 584 Homework 2

Avitej Iyer

A look at balancing a double pendulum using reinforcement learning

October 26, 2024

Github repository referenced : *Control of an Inverted Double Pendulum using Reinforcement Learning* by Fredrik K. Gustafsson
Project description page by author : https://www.fregu856.com/project/cs229/

Abstract

This repository implements reinforcement learning (RL) to control an inverted double pendulum on a cart using Q-learning with a linear function approximation. The inverted pendulum problem is a classic benchmark in control theory and RL, where the objective is to balance the pendulum in its unstable upright position by applying forces to the cart. This is also a great application of machine learning in an otherwise purely physics-based or engineering problem. The system is simulated in a custom environment that models the physical dynamics of the cart and pendulum, such as position, velocity, and angles.

The RL agent learns to balance the pendulum by interacting with the environment, where it receives rewards based on how well it maintains the pendulum's balance. The core algorithm used is Q-learning, which updates the Q-values associated with state-action pairs iteratively. However, due to the complexity of the continuous state space, the Q-function is approximated using a linear function rather than a full table of state-action pairs. This function approximation reduces the memory and computational requirements of the learning process.

The Q-values are updated using the Bellman equation, which takes into account the immediate reward and the estimated future rewards. The policy for selecting actions is based on an epsilon-greedy strategy, where the agent occasionally chooses random actions to explore new possibilities, but mainly exploits actions that have been shown to maximize rewards.

While the RL agent succeeds in balancing the pendulum, it fails to learn the more complex swing-up maneuver, highlighting the limitations of Q-learning with linear approximators for highly nonlinear control tasks. The repository's code also includes discussions and visualizations of the results, along with suggestions for improving the performance in more challenging control problems.

Project Process Overview

- **Initialize the Environment:** A custom simulation of the double inverted pendulum is created, modeling physical dynamics like position, velocity, and angle of the pendulum and cart. The environment outputs these states for further use.
- **Define the State and Action Space:** The state includes variables such as cart position and velocity, and pendulum angles and angular velocities. The actions involve applying a force to the cart, which affects the pendulum's behavior.
- **Q-Learning with Function Approximation:** A linear function approximator is used to estimate Q-values for each state-action pair. This makes learning feasible in continuous state spaces.
- **Policy Execution:** An epsilon-greedy policy is used to select actions—exploiting the best-known actions most of the time while exploring new ones occasionally.
- **Q-Value Update:** After each action, the Q-value is updated using the Bellman equation, considering the immediate reward and the estimated future rewards.
- **Learning and Iteration:** The agent iteratively interacts with the environment, refining the policy through updates to the Q-function.
- **Evaluation and Limitations:** The performance of the trained agent is evaluated. While balancing is achieved, the swing-up maneuver proves too complex due to the limitations of Q-learning with linear approximation.

Explanation of Code

For the code itself, the project doesn't use standard machine learning libraries like TensorFlow or Pytorch. Instead, it relies on manual implementation of the reinforcement learning algorithm. All of the computations, such as dot products for Q-value estimation and weight updates, are performed using NumPy for matrix operations. This manual approach offers flexibility and transparency for understanding the underlying reinforcement learning mechanics without the abstractions that come with larger ML libraries.

Presented below are the functions I consider most important to the Q-learning application:

- **q_hat():** Estimates Q-values, essential for decision-making.
- **phi_critic():** Encodes state-action pairs into feature vectors for approximation.
- **get_action():** Handles action selection based on the current policy.
- **update_weights():** Performs the critical Q-learning weight update using the Bellman equation.
- **mu():** Computes the expected action based on the policy.

Below is a more in-depth look at each of the functions above :

- Function q_hat() :

```
# This function estimates the Q-value for a given state-action
# pair using a linear function approximator.
# It takes in the current state, the action taken, and the
# weights (w), which represent the learned model.
# The function returns the estimated Q-value by computing
# the dot product between the feature vector and the weights.

def q_hat(state, action, w):
    # Generate a feature vector based on the state and action.
    X = phi_critic(state, action)
    # Calculate the Q-value by performing a dot product with weights.
    output = np.dot(X, w)
    # Return the estimated Q-value.
    return output
```

- Function phi_critic() :

```
# This function creates a feature vector that represents the state-action
↪  pair.
# It takes in the current state and action, and returns a feature vector,
# which is a numerical representation used by the Q-function approximator.

def phi_critic(state, action):
    # Example feature vector combining state and action.
    return np.array([state[0], state[1], action])
```

- Function get_action() :

```
# This function selects an action for the agent based on the current policy.
# It takes in the current state and the weights of the policy (v) and
↪  computes the mean action.
# The action is sampled from a normal distribution centered at this mean,
↪  with some exploration.
# The action is then clipped to stay within the allowed range and returned.

def get_action(state, v):
    # Compute the mean action based on the state and policy.
    mean = mu(state, v)
    # Sample an action with added exploration (randomness).
    action = np.random.normal(mean, sigma)

```

```
    # Clip the action to the upper limit.
    if action > 40:
        action = 40
    # Clip the action to the lower limit.
    elif action < -40:
        action = -40

    # Return the selected action.
    return action
```

- Function update_weights():

```
# This function updates the Q-function weights using the Q-learning algorithm.
# It takes in the current state, action, reward, next state, and a boolean indicating whether the episode is done.
# The function calculates the TD error based on the difference between the current Q-value and the target Q-value,
# then updates the weights using the learning rate and the TD error.

def update_weights(state, action, reward, next_state, done):
    # Calculate the current Q-value for the state-action pair.
    q_value_current = np.dot(weights, state)
    # Estimate the Q-value for the next state.
    q_value_next = np.dot(weights, next_state)

    # Compute the target Q-value using the Bellman equation.
    target = reward + (1 - done) * discount_factor * q_value_next
    # Calculate the temporal difference (TD) error.
    td_error = target - q_value_current

    # Update the weights using the TD error and learning rate.
    # Doesn't return anything because this function directly modifies the weights global variable
    weights += learning_rate * td_error * state
```

- Function mu():

```
# This function computes the mean action for a given state using the policy.
# It takes in the current state and the policy weights (v) and returns the mean action
# by calculating the dot product between the state features and the policy weights.

def mu(state, v):
```

```
    # Compute the action using a linear function of the state and policy weights.
    return np.dot(state, v)
```

Conclusion

This assignment provided an in-depth look at the implementation of reinforcement learning through Q-learning with linear function approximation for controlling an inverted double pendulum. By analyzing key functions and their interconnections, I explored how the agent learns to balance the pendulum in a complex environment. This project highlights the strengths and limitations of function approximation in RL, offering valuable insights into real-world applications of machine learning in control tasks.