6-2 Assignment: Cartpole Revisited

Joel De Alba

Southern New Hampshire University

Professor Tim Alexander

08 / 19 / 23

The initial code provided for the Cart-Pole issue employed a Q-learning algorithm, which is a value-based approach to reinforcement learning. In value-based algorithms such as Q-learning, the agent acquires the ability to approximate the value of various states or state-action pairings, and subsequently use these value approximations to optimize its expected rewards. The Q-table was employed as a data structure to record the estimated values associated with state-action pairings. The Q-values were modified by the agent in accordance with the rewards obtained, and subsequently utilized to determine the optimal action to be taken in each state.

The REINFORCE algorithm, a policy-based methodology, focuses on optimizing the policy parameters in order to maximize the anticipated return. Rather than making approximations of values, the emphasis is placed on acquiring knowledge about the optimal policy or approach that leads to the attainment of greater rewards. The policy is subsequently enhanced by using gradient information in order to enhance its performance over a period of time. The cartpole problem pertains to the task of maintaining equilibrium of a pole positioned atop a cart in motion. The objective is to implement measures that maintain the vertical stability of the pole and mitigate the risk of it toppling over.

The subsequent section outlines a straightforward, sequential procedure for executing the REINFORCE algorithm in order to address the Cartpole problem.

1. Start by importing the necessary libraries. The utilization of NumPy is essential for doing numerical operations, while the Gym library is required for the creation and interaction with the CartPole environment.
2. Initialize Environment by creating the CartPole environment using gym.make('CartPole-v1'). By utilizing this functionality, users will be able to obtain access to many qualities such as the status space, action space, and other relevant information pertaining to the environment.
3. Initialize the policy parameters randomly. In the present scenario, we are dealing with a straightforward parameterization comprising four state features, namely the position and velocity of the cart, as well as the angle and angular velocity of the pole. Additionally, there are two available actions, namely pushing the cart to the left or to the right.
4. Set the Learning Rate (learning\_rate) to control the step size during parameter updates.
5. Specify the number of episodes (num\_episodes) you want the algorithm to run for.
6. REINFORCE Algorithm:
   1. The REINFORCE algorithm involves iterating through each episode from 1 to the specified number of episodes (num\_episodes).
   2. Begin by initializing empty lists to use as containers for episode rewards and gradients.

Episode Loop:

* 1. Reset the environment to its initial state and initialize done as False.
  2. While the episode is not done:

d(a). Compute the probabilities of actions by utilizing the existing policy parameters. The dot product between the state and parameters is transformed into probabilities and subsequently normalized.

d(b). Select an action by utilizing the np.random.choice function in accordance with the given probabilities.

d(c). Execute the selected course of action inside the given environment and record the subsequent state (next\_state), the reward obtained (reward), and if the episode has concluded (done).

d(d). Calculate the gradient of the log probability of the chosen action with respect to the policy parameters. This gradient is an outer product of the state and action probabilities.

1. Calculate Episode Total Reward: Sum up the rewards obtained in the current episode to get the total reward (total\_reward).
2. Update Policy Parameters: For each time step within the episode:
   1. Calculate the discounted future rewards (discount\_factor) for the remaining time steps using the discount factor (gamma).
   2. Update the policy parameters using the REINFORCE update rule: parameters += learning\_rate \* discount\_factor \* gradient.
3. Print Episode Information: After each episode, print the episode number, total reward, and any other relevant information.
4. Close Environment: After all episodes are completed, close the CartPole environment using env.close().

The following is the pseudocode for the REINFORCE algorithm implementation for the cartpole problem:

# Import necessary libraries

import numpy as np

import gym

# Initialize environment

env = gym.make('CartPole-v1')

# Initialize policy parameters randomly

policy\_param = np.random.rand(4, 2) # 4 state features, 2 possible actions

# Set learning rate

learning\_rate = 0.01

# Number of episodes

num\_episodes = 1000

# REINFORCE algorithm

for episode in range(num\_episodes):

episode\_rewards = []

episode\_gradients = []

state = env.reset()

done = False

while not done:

# Choose an action based on the policy

action\_probs = np.exp(np.dot(state, policy\_param)) # Convert to probabilities

action\_probs /= np.sum(action\_probs) # Normalize

action = np.random.choice(np.arange(len(action\_probs)), p=action\_probs)

# Take the chosen action and observe next state and reward

next\_state, reward, done, \_ = env.step(action)

# Calculate the gradient of the log probability of chosen action

gradient = np.outer(state, action\_probs)

episode\_rewards.append(reward)

episode\_gradients.append(gradient)

state = next\_state

# Calculate the total reward for the episode

total\_reward = sum(episode\_rewards)

# Update policy parameters using the REINFORCE update rule

for t in range(len(episode\_rewards)):

discount\_factor = sum([gamma\*\*i \* episode\_rewards[i] for i in range(t, len(episode\_rewards))])

policy\_param += learning\_rate \* discount\_factor \* episode\_gradients[t]

# Print episode information

print(f"Episode {episode+1}/{num\_episodes} - Total Reward: {total\_reward}")

# Close the environment

env.close()

There are three main types of approaches to reinforcement learning. Value-Based approaches, Policy-Based approaches, and Module-Based. Hybrid approaches are also explained to illustrate how actor-critic models work.

In value-based approaches, the agent learns to estimate the value of being in a particular state or taking a specific action. The value represents the expected cumulative reward the agent can achieve from that state onwards by following a specific strategy. Algorithms such as Q-learning and Deep Q-Networks (DQN) fall under this category.

Policy-based approaches directly learn the policy itself without estimating value functions. The goal is to find the optimal policy that directly maps states to actions in a way that maximizes the expected cumulative reward. Policy-based approaches are well-suited for both discrete and continuous action spaces, making them especially effective in complex and continuous environments. REINFORCE and Proximal Policy Optimization (PPO) algorithms fall under this category.

A Module-Based approach, also known as a modular approach, is a software development methodology that involves breaking down a complex system or application into smaller, self-contained units called modules. Each module represents a distinct functionality or feature of the system, and these modules can be developed, tested, and maintained independently.

Hybrid approaches, like actor-critic methods combine elements of both value-based and policy-based approaches where the agent consists of two components: an actor that learns the policy and a critic that estimates the value function. This combination allows for more stable and efficient learning, as the critic provides feedback to the actor on the quality of the chosen actions.

The following image depicts the RL approaches explained above:

A diagram of a model based method

Description automatically generated

Image provided by Synopsys at: Reinforcement learning explained in 90 seconds. (2021, April 27). [Video]. https://www.synopsys.com/ai/what-is-reinforcement-learning.html

Actor-Critic aims to address the limitations of pure policy-based or value-based approaches by leveraging the strengths of each. In an Actor-Critic setup, an agent consists of two main components: the actor and the critic. The actor's role is to learn and improve the policy. It is responsible for selecting actions in the environment based on the current state. The actor aims to find the best policy that directly maps states to actions, maximizing the expected cumulative reward. The actor then guides the agent to make better decisions in the environment and policy updates are driven by the reward received from the environment.

The critic focuses on estimating the value of states or state-action pairs. Its role is to provide feedback to the actor by evaluating how good the chosen actions are. The critic helps the actor make faster and more informed decisions by estimating the expected future rewards. Value-based methods, such as Q-learning or Deep Q-Networks, are often used as the basis for the critic component.

The step-by-step process to implement the actor-critic algorithm for the cartpole problem is as follows:

1. Create and initialize the actor and critic neural network models. The actor model learns to select actions, and the critic model estimates state values.
2. Define and set Hyperparameters like learning rates, discount factor (gamma), number of steps per episode, and maximum episodes.
3. Repeat Until Convergence:

a. Execute the current policy to generate a trajectory (sequence of states, actions, rewards).

b. Calculate the total reward achieved from the trajectory.

c. Update Critic Model by computing the temporal difference error using the total reward and the estimated state values from the critic. The critic model's parameters are updated to reduce this error.

d. Calculate the advantage function, which quantifies how much better an action is compared to the average action in a given state.

e. Update Actor Model by computing the gradient of the policy objective using the advantage function and the policy gradient theorem. The actor model's parameters are updated to improve the policy based on the reward signal.

f. Repeat these steps until convergence criteria are met (e.g., a certain number of episodes or a satisfactory level of performance).

Here is the pseudocode for implementing the Actor-Critic algorithm for solving the Cartpole problem:

# Initialize actor and critic neural network models

initialize\_actor\_and\_critic\_models()

# Define hyperparameters

learning\_rate\_actor = 0.001

learning\_rate\_critic = 0.01

gamma = 0.99

num\_steps\_per\_episode = 200

max\_episodes = 1000

convergence\_threshold = 200

# Repeat until convergence

for episode in range(max\_episodes):

state = reset\_environment()

total\_reward = 0

for step in range(num\_steps\_per\_episode):

# Choose an action using the actor's policy

action = select\_action\_with\_actor(state)

# Take the chosen action in the environment

next\_state, reward, done = take\_action\_in\_environment(action)

# Update total reward

total\_reward += reward

# Compute TD error for the critic update

td\_error = compute\_td\_error(reward, gamma, state, next\_state, done)

# Update the critic model

update\_critic\_model(td\_error)

# Calculate the advantage function

advantage = calculate\_advantage\_function(td\_error, state)

# Update the actor model using the advantage and policy gradient

update\_actor\_model(advantage)

# Move to the next state

state = next\_state

if done:

break

# Print progress

print(f"Episode {episode+1} - Total Reward: {total\_reward}")

# Check for convergence

if total\_reward >= convergence\_threshold:

print("Convergence reached!")

break

# Apply the learned policy

apply\_learned\_policy()

Here is an image that depicts the functionality of the A2C algorithm, this image was provided by the following source:

Karagiannakos, S. (2018b, November 17). The idea behind Actor-Critics and how A2C and A3C improve them | AI Summer. AI Summer. https://theaisummer.com/Actor\_critics/

A diagram of a problem

Description automatically generated

Sources

Reinforcement learning explained in 90 seconds. (2021, April 27). [Video]. <https://www.synopsys.com/ai/what-is-reinforcement-learning.html>

Karagiannakos, S. (2018, November 17). The idea behind Actor-Critics and how A2C and A3C improve them | AI Summer. AI Summer. <https://theaisummer.com/Actor_critics/>

Juliani, A. (2018, June 21). Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C). Medium. <https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2>

Emergent tool use from multi-agent interaction. (n.d.). <https://openai.com/research/emergent-tool-use>

OpenAI. (2019b, September 17). Multi-Agent hide and seek [Video]. YouTube. <https://www.youtube.com/watch?v=kopoLzvh5jY>

Part 2: Kinds of RL Algorithms — Spinning Up  documentation. (n.d.). <https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#a-taxonomy-of-rl-algorithms>

Yoon, C. (2021, December 7). Deriving policy gradients and implementing REINFORCE. Medium. <https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63>

Gulli, A., & Pal, S. (2017). *Deep learning with keras : Get to grips with the basics of keras to implement fast and efficient deep-learning models*. Packt Publishing, Limited.

Beysolow, I. T. (2019). *Applied reinforcement learning with python : With openai gym, tensorflow, and keras*. Apress L. P..

Lamba, A. (2018b, September 3). An introduction to Q-Learning: reinforcement learning. Medium. <https://medium.com/free-code-camp/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc>

Düğmeci, D. (2022, July 6). Finding Shortest Path using Q-Learning Algorithm - Towards Data Science. Medium. <https://towardsdatascience.com/finding-shortest-path-using-q-learning-algorithm-1c1f39e89505>

Surma, G. (2021, October 13). Cartpole - Introduction to Reinforcement Learning (DQN - Deep Q-Learning). Medium. <https://gsurma.medium.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288>

Manrique Escobar, C. A., Pappalardo, C. M., & Guida, D. (2020). A parametric study of a deep reinforcement learning control system applied to the swing-up problem of the cart-pole. Applied Sciences, 10(24), 9013.

Kumar, S. (2020). Balancing a CartPole System with Reinforcement Learning--A Tutorial. arXiv preprint arXiv:2006.04938.

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8, 279-292.

Dearden, R., Friedman, N., & Russell, S. (1998). Bayesian Q-learning. Aaai/iaai, 1998, 761-768.

Aleksandar Haber. (2023, February 5). Introduction to OpenAI Gym (Gymnasium): Cart-Pole Environment - Reinforcement Learning Tutorial [Video]. YouTube. <https://www.youtube.com/watch?v=2sp_eucoX2I>

Admin. (2023, January 31). Cart Pole Control environment in OpenAI Gym (Gymnasium)- Introduction to OpenAI Gym – fusion of engineering, control, coding, machine learning, and science. <https://aleksandarhaber.com/cart-pole-control-environment-in-openai-gym-gymnasium-introduction-to-openai-gym/>

Admin. (2023b, February 10). Deep Q Networks (DQN) in Python from scratch by using OpenAI GyM and TensorFlow- Reinforcement Learning tutorial – fusion of engineering, control, coding, machine learning, and science. <https://aleksandarhaber.com/deep-q-networks-dqn-in-python-from-scratch-by-using-openai-gym-and-tensorflow-reinforcement-learning-tutorial/>