Reinforcement Learning Project

Yana Holovatska

Problem 1

Summary of problem

The objective of the Cartpole problem is to develop an agent capable of balancing a pole on a moving cart for as long as possible. The agent must learn to control the cart's position on a track by applying forces either to the left or right, in this way adjusting the pole's position to prevent it from falling.

Environment Configuration

The environment defines several physical parameters:

- 'self.gravity' defines the gravity constant
- 'self.masscart = 1.0' and 'self.masspole = 0.1' define the mass of cart and pole.
- 'self.length', the length of the pole
- 'self.force_mag', the force magnitude applied at each step
- 'self.tau' - seconds in-between updates

Environment Boundaries

The episode termination conditions are defined based on the cart's horizontal displacement

```
    self.x_threshold = 2.4
```

and the pole's angle with respect to vertical

```
    self.theta_threshold_radians = 12 * 2 * math.pi / 360
```

These thresholds help in defining the episode's end by tracking if the pole falls down or the cart moves out of the allowed horizontal limit.

State Space

- **Components of state**: The state consists of four continuous variables: the cart's position x, cart's velocity x_dot, pole's angle $theta$, and the pole's angular velocity $theta\_dot$.
- **Observation space**: Defined using spaces.Box, which specifies the allowable high and low values for each component in the state. This encapsulates the possible range of values each element of the state can take.

Action Space

- **Discrete actions**: The environment supports two discrete actions (spaces.Discrete(2)): pushing the cart to the left (0) and to the right (1). These actions directly apply force to the cart (with the magnitude self.force_mag), which in turn affects the system's dynamics.

Dynamics and Transitions

- **Force Application**: Based on the action taken, a force is applied either to the left or right, impacting the cart's acceleration and subsequently the pole's dynamics.
- **Physics Calculations**: Detailed calculations for the pole's angular acceleration and the cart's acceleration are performed using trigonometric functions and Newtonian mechanics principles.
- **Integration Methods**: The state is updated using an Euler or semi-implicit Euler method, defining how the cart's position and velocity, along with the pole's angle and angular velocity, are updated over discrete time steps self.tau.

Rewards and Episode Termination

- **Reward Structure**: A reward of 1.0 is given for each time step the episode continues without termination. This motivates the agent to keep the pole upright and the cart within bounds.
- **Termination Conditions**: The episode terminates if the cart moves out of the horizontal boundaries or if the pole falls beyond the specified angle threshold. Additionally, there's a step limit (500 steps), after which the episode also terminates.

Reset and Rendering

- **Reset Function**: Resets the environment to a starting state with small variations in the initial conditions.
- **Rendering**: Provides a visual representation using Pygame. It visualizes the cart's position and the pole's status relative to the environment's dimensions.

Problem 2

I have changed them as following:

```
    # ---> TODO: change input and output sizes depending on the
  environment
    input_size = 4
    nb_actions = 2
```

Input size is 4, since vector which describes state is 4-dimensional:

- cart position
- cart velocity
- angle of the pole
- angular velocity of the pole

Output size is 2, since there are two actions, that actor can perform:

- apply force in left direction
- apply force in right direction

Selecting action

```
    # ---> TODO: how to select an action
    action = torch.argmax(probabilities).item()
```

To select an action, we simply use argmax function, to find the action with the highest probability

Performance

```
(env) yana@Yanas-MBP to_students % python3 ./1_test_model.py
ActorModelV0(
  (fc0): Linear(in_features=4, out_features=128, bias=True)
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (policy): Linear(in_features=128, out_features=2, bias=True)
)
total_reward = 10.0
```

Figure 1: Output of the script

As we see on Figure 1, the model lasted only 10 time units, getting reward of 10. Episode terminated because the pole has fallen.

Such a poor performance is due to the fact, that we haven't trained the policy network. We will do this in the next problem to improve performance

Problem 3

Subproblem 1

The discount factor β is a parameter in reinforcement learning that determines the present value of future rewards. It is a number between 0 and 1 and influences how much the agent values immediate rewards compared to future rewards.

A higher discount factor means that future rewards are valued more highly, encouraging the agent to consider long-term outcomes. Conversely, a lower discount factor makes the agent prioritize immediate rewards, as future rewards contribute less to the total expected return.

The following code computes the discounted reward.

```
    # Compute discounted sum of rewards
    # ---------------------------------------
    # Current discounted reward
    discounted_reward = 0.0

    # List of all the discounted rewards, for each time step
    discounted_rewards = list()

    # ---> TODO: compute discounted rewards
    for reward in reversed(saved_rewards):
        discounted_reward *= DISCOUNT_FACTOR
        discounted_reward += reward
        #inserting in the beginning to bring rewards back in
  order
```

```
        discounted_rewards.insert(0, discounted_reward)
```

Subproblem 2

Reinforcement loss

We will use a Monte-Carlo estimate over one episode:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{\infty} G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

It is possible to transform equation by putting G_t inside the nabla operator:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{\infty} \nabla_\theta G_t \cdot \log \pi_\theta(a_t|s_t)$$

Now we can compute this inside the loop, where on each iteration we calculate $G_t \cdot \log \pi_\theta(a_t|s_t)$.

```
        # For each time step
        actor_loss = list()
        for p, g in zip(saved_probabilities, discounted_rewards):

            # ---> TODO: compute policy loss
            time_step_actor_loss = - g * torch.log(p)

            # Save it
            actor_loss.append(time_step_actor_loss)
```

We additionally add minus, so that we maximize the reward, rather than minimizing it.

Subproblem 3

Choosing parameters

The learning rate is a hyperparameter that determines the size of the steps taken during the process of gradient descent (in our case ascent). It controls how much the parameters (weights in neural networks) are adjusted with respect to the gradient of the loss function.

- High Learning Rate: Can cause the training process to converge very quickly or potentially overshoot the minimum, resulting in divergence or unstable training.
- Low Learning Rate: Ensures more stable and precise convergence but can make the training process slower and sometimes get stuck in local minima.

I've tried different learning rate parameters

- values 0.1 and 0.01 both provided unstable training. Algorithms was getting stuck between rewards 8-10.
- value 0.001 was providing reliable training, and converged in about 400 iterations.
- value 0.0001 took much longer time, than previous one.

Thus, I conclude that we might use **0.001** as learning rate.

Also, i have tried different values for discount factor between 0.9 and 0.99, but it didn't give much difference. We will just stick with **0.99** value

Subproblem 4

Stopping training

I have set the training to stop, when algorithm achieves 10 consecutive rewards of HORIZON value (maximum possible reward in environment of that length). For this I calculate rolling average, and check in the end of each iteration whether stop condition is satisfied.

This ensures, that we stop the training when the algorithm has converged to 500, and stabilized around that value.

Define this before While True loop:

```
    ROLLING_WINDOW_SIZE = 15
    rolling_rewards = []

    def should_stop_training(rolling_rewards, threshold):
        if len(rolling_rewards) >= ROLLING_WINDOW_SIZE:
            average_reward = sum(rolling_rewards) / len(
  rolling_rewards)
            return average_reward >= threshold
        return False
```

I have tried different values for ROLLING_WINDOW_SIZE, and found 15 to work the best. With 10, it often performed good in training, but was getting rewards of around 200 during testing (because cart was going out of frame, while trying to balance pole). Values higher than 15 were taking more time during training.

Add this inside the loop, after the reward of episode has been calulated:

```
        # Episode total reward
        episode_total_reward = sum(saved_rewards)

        # ---> TODO: when do we stop the training?
        rolling_rewards.append(episode_total_reward)

        # Maintain the size of rolling_rewards to be exactly
  ROLLING_WINDOW_SIZE
        if len(rolling_rewards) > ROLLING_WINDOW_SIZE:
            rolling_rewards.pop(0)  # Remove the oldest reward
```

Lastly, add this inside the loop in the very end:

```
if should_stop_training(rolling_rewards, HORIZON):
    print(f"Training has converged with rolling average: {sum(
    rolling_rewards) / len(rolling_rewards):.2f} in {
    training_iteration} iterations")
    break
```

Results

```
(env) yana@Yanas-MBP to_students % python3 ./2_reinforce.py
ActorModelV0(
  (fc0): Linear(in_features=4, out_features=128, bias=True)
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (policy): Linear(in_features=128, out_features=2, bias=True)
)
iteration 50 - last reward: 68.00
iteration 100 - last reward: 180.00
iteration 150 - last reward: 205.00
iteration 200 - last reward: 160.00
iteration 250 - last reward: 500.00
iteration 300 - last reward: 150.00
iteration 350 - last reward: 500.00
Training has converged with rolling average: 500.00 in 391 iterations
```

Figure 2: Training results

```
(env) yana@Yanas-MBP to_students % python3 ./3_test_model.py
ActorModelV0(
  (fc0): Linear(in_features=4, out_features=128, bias=True)
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (policy): Linear(in_features=128, out_features=2, bias=True)
)
total_reward = 500.0
```

Figure 3: Test results

Problem 4

It doesn't converge anymore, because environment stopped being Marcovian. The information about current state, such as position of cart and angle of pole is not enough to predict future states.

Idea

However, we can make environment Marcovian again without adding velocities.
We can simply include in the vector information about previous state. We know that time between states is constant, and set to self.tau. Thus, information about cart and pole velocity can be inferred (because both time and displacement is known). Thus, this information would be enough to make environment Markovian.

Modify code

First, we add line to reset function, which will initialize previous state as zero vector.

```
    self.previous_state = np.array([0,0,0,0], np.float32)
```

Listing 1: Adding line in reset function

Then, we are adding another line in step function, which copies current state before it will be updated.

```
    self.previous_state = self.state.copy()
```

Listing 2: Adding line in step function

Lastly, we add this line to process_state function.

```
    # Include cart position and pole angle from previous position
    to working state
    processed_state = np.array([self.state[0],
                                self.state[2],
                                self.previous_state[0],
                                self.previous_state[2]])
```

Listing 3: Adding line in process_state function

Now, our input size is again 4, so we modify the input size:

```
    input_size = 4
    nb_actions = 2
```

Listing 4: Modify actor_v1 variables

Result

```
iteration 5150 - last reward: 500.00
iteration 5200 - last reward: 500.00
Training has converged with rolling average: 500.00 in 5232 iterations
(env) yana@Yanas-MBP to_students % python3 ./3_test_model.py
ActorModelV0(
  (fc0): Linear(in_features=4, out_features=128, bias=True)
  (fc1): Linear(in_features=128, out_features=128, bias=True)
  (policy): Linear(in_features=128, out_features=2, bias=True)
)
total_reward = 500.0
```

Figure 4: Training and testing results

We see that now it converges again. However, it notably takes much more time to converge. While using velocities, it took about 400 iterations, but now it converged in 5000 iterations.