ENGR 520 Homework 4

Name: Anthony Su
Date: May 27, 2025

CODE: `https://github.com/exurl/ENGR520`

General References:

- docs: `gymnasium.env` `https://gymnasium.farama.org/api/env/#gymnasium.Env.reset`
- docs: `gymnasium` `CartPole` environment: `https://gymnasium.farama.org/environments/classic_control/cart_pole/`
- docs: OpenAI "Spinning Up" deep RL tutorial: `https://spinningup.openai.com/en/latest/`

Q Learning

Q Learning References

- blog: Q-learning pseudocode: `https://rezaborhani.github.io/mlr/blog_posts/Reinforcement_Learning/Q_learning.html#3.2--The-basic-Q-Learning-algorithm`
- blog: Deep Q-Learning Algorithm: `https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-algorithm`

Q Learning Notes

- Built-in `gymnasium` environments cannot have their state limits modified
  - in the `CartPole` environment, the there are no limits on cart velocity or pole rotation rate.
  - the `state_bounds` variable I define defines the bounds of the state discretization bins, not the actual allowed state limits.
- Observation space: 4-element array of (cart position, cart velocity, pole angle, pole angular velocity)
  - Sample with `env.observation_space.sample()`
  - The observation space is bucketed into discrete sub-spaces. Each sub-space has a corresponding entry in the Q table. Action selection function: Q table
  - Action space: boolean (push left or push right)
  - Epsilon-greedy exploration is used in action selection function with decaying exploration rate
- The Q table is 5-dimensional. The sizes of its dimensions are defined in the variable `num_buckets` with an additional final dimension of 2 (which represents the decision probabilities of left force and right force, respectively).
  - Note that since our buckets do not span the allowed observation space, there is no distinction between behavior at cart speed (or pole angular velocity) of 4 or 4 trillion.
- **epsilon-greedy**: ϵ chance of making random move instead of policy move during training. this causes exploration.
  - decrease ϵ over time during training to increase exploitation towards the end of training

alternative method: deep Q learning, where Q distribution is a neural network with continuous inputs and outputs

see pseudocode below

Proximal Policy Optimization

References

- docs: OpenAI "Spinning Up" policy optimization tutorial: `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`

PPO Notes

- **PPO**: "proximal" because updates to policy are clipped to stay within a certain distance (ratio-wise) of the original policy.

    - This prevents overshoots and increases stability.
- **Actor-Critic**: One network for policy $\pi(s, a)$ (actor) and one network for estimating state value $V(s)$ (critic).
- **return**: the realized (not predicted) value of a state
- **advantage**: delta between actual and expected return of a state
    - used to compute gradient for training
- **rollout**: a time series fragment of the last _ time steps
- **bootstrapping**: in this context, bootstrapping is approximating future rewards as the critic network's value estimate
    - this is done if the policy is updated when not at the end of a rollout
- **entropy**: information theory definition of entropy
    - used to encourage less confident predictions during training, which increases exploration
- `torch.distributions.Categorical` is a discrete probability distribution for a finite set of known options

see pseudocode below

Exercises

i. Definitions

- Reward $r(s)$: the feedback signal from the environment which indicates that a desirable state has been reached
- Value $V_\pi(s)$: the expected total remaining reward of enacting a known policy π for the rest of the episode starting from $s_0=s$
- Quality $Q(s, a)$: the expected value of s if action a is taken and all subsequent future actions are optimal.

ii. Q-Learning Pseudocode

Q-Learning with Table

1. Set hyperparameters and iteration parameters
2. Initialize environment and action selection function (e.g. Q table).
3. Get environment observation
4. Call action selection function (Q table) to determine action
5. Take action and time step; observe new state and classify which bin it belongs to (if the state is originally continuous).
6. Update Q table using the Bellman equation
   $Q_{\text{new}}(s_t, a_t) = Q_{\text{old}}(s_t, a_t) + \alpha\left(r(s_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)$
7. Repeat steps 3-6 for each time step in the episode.
8. Repeat steps 3-7 for each episode.

Q-Learning with Neural Network

1. Set hyperparameters and iteration parameters.
2. Initialize environmentm, main deep Q network, target deep Q network
3. Initialize a replay buffer.
4. Start training loop for a specified number of episodes.
5. For each episode, reset the environment and get the initial state.
6. Start timestep loop for the episode.
7. Select an action using an epsilon-greedy policy based on the main Q-network's output for the current state.
8. Take the selected action in the environment; observe the next state, reward, and terminal status.
9. Store the transition (current state, action, reward, next state, terminal status) in the replay buffer.
10. Periodically perform a policy update:
    - Sample a random batch of transitions from the replay buffer.
    - Calculate target Q-values for the batch using the target Q network (reward + discounted Q from next state).
    - Calculate current Q-values for the batch using the main Q network.
    - Calculate the loss (MSE) between the current Q-values and the target Q values for the batch.
    - Perform a gradient descent step to update the weights of the main Q network based on the loss.
    - Periodically update the target Q network by copying the weights from the main Q network.

11. Repeat steps 7-11 for each time step in the episode.
12. Repeat steps 5-12 for each episode.

iii. Proximal Policy Optimization Pseudocode

1. Set hyperparameters
2. Initialize environment, policy network (actor-critic), old policy network, and rollout buffer.
3. Start training loop for a specified number of episodes.
4. For each episode, reset the environment and get the initial state.
5. Start rollout collection loop
6. Use the old policy network to select an action based on the current state, get log probability and state value.
7. Store state, action, log probability, and state value in the buffer.
8. Take the selected action in the environment; observe the next state, reward, and terminal status.
9. Store the reward and terminal status in the buffer.
10. Update current state.
11. Repeat steps 5-10 until the rollout is complete (buffer size reaches `update_timestep` or episode ends).
12. Periodically perform a policy update:
    - Calculate returns for the collected rollout data.
    - Calculate advantages.
    - Optimize the policy network for a set number of epochs:
        - Evaluate the current policy on the rollout data to get new log probabilities, state values, and entropy.
        - Calculate the ratio of new/old log probabilities.
        - Compute the PPO-Clip loss.
        - Compute the value loss (e.g., MSE between current state values and returns).
        - Compute the entropy bonus.
        - Calculate the total loss (PPO-Clip loss + Value loss - Entropy bonus).
        - Perform a gradient descent step to update the policy network parameters.
    - Copy the weights from the policy network to the old policy network.
    - Clear the rollout buffer.
13. Repeat steps 4-12 for each episode.