Agent Playground Solution

> 机器学习概论lab4
>
> **Author:**@Rosykunai
>
> **Date:**2024 年 12 月

Agent Playground Solution

1. Value Iteration

1-(a) computeQ

```
def computeQ(V: Dict[StateT, float], state: StateT, action: ActionT) ->
float:
        # Return Q(state, action) based on V(state)

        # BEGIN_YOUR_CODE
        return sum([prob * (reward + discount * V[nextState]) for
nextState, prob, reward in succAndRewardProb[(state, action)]])
        # END_YOUR_CODE
```

1-(b) computePolicy

```
def computePolicy(V: Dict[StateT, float]) -> Dict[StateT, ActionT]:
        # Return the policy given V.
        # Remember the policy for a state is the action that gives the
greatest Q-value.
        # IMPORTANT: if multiple actions give the same Q-value, choose the
largest action number for the policy.
        # HINT: We only compute policies for states in stateActions.

        # 1-a
        # BEGIN_YOUR_CODE
        pi = {}
        for state, actions in stateActions.items():
            pi[state] = max(actions, key=lambda action: computeQ(V, state,
action))
        return pi
        # END_YOUR_CODE
```

1-(c) valueIteration

```
# 1-c
# BEGIN_YOUR_CODE
for state in stateActions.keys():
    newV[state] = max([computeQ(V, state, action) for action in
stateActions[state]])
if all([abs(newV[state] - V[state]) < epsilon for state in
stateActions.keys()]):
    break
# END_YOUR_CODE
```

2. Model-based Monte Carlo

2-(a) getAction

```
# 2-a
# BEGIN_YOUR_CODE
if not explore:
    explorationProb = 0.0
if state not in self.pi or random.random() < explorationProb:
    return random.choice(self.actions)
return self.pi[state]
# END_YOUR_CODE
```

2-(b) incorporateFeedback

```
# 2-b
# BEGIN_YOUR_CODE
for (state, action), nextStates in self.tCounts.items():
    for nextState, count in nextStates.items():
        transitionProb = count / sum(nextStates.values())
        expectedReward = self.rTotal[(state, action)][nextState] / count
        succAndRewardProb[(state, action)].append((nextState,
transitionProb, expectedReward))
self.pi = valueIteration(succAndRewardProb, self.discount)
# END_YOUR_CODE
```

3. Q-learning

3-(a) getAction

```
# 3-a
# BEGIN_YOUR_CODE
explorationProb = explorationProb if explore else 0.0
options = [(k[1], v) for k, v in self.Q.items() if k[0] == state]
if random.random() < explorationProb or not options:
    return random.choice(self.actions)
return max(options, key=lambda x: x[1])[0]
# END_YOUR_CODE
```

3-(b) incorporateFeedback

```
# 3-b
# BEGIN_YOUR_CODE
target = reward
if not terminal:
    target += self.discount * max([self.Q[nextState, nextAction] for
nextAction in self.actions])
self.Q[state, action] += self.getStepSize() * (target - self.Q[state,
action])
# END_YOUR_CODE
```

4. [Optional] Reinforce

4-(a) sample traj

```
# 4-a
# Sample an episode using Gymnasium environment and Policy Network.
# Use policy.getAction(state) to get the action and log probability.
# Use env.step(action) to get the next_state, reward, terminated,
truncated and info.
# You can look up the MountainCarMDP.transition() function as a reference.
# BEGIN_YOUR_CODE

for t in range(max_t):
    state = torch.from_numpy(state).float().unsqueeze(0)
    action, log_prob = policy.getAction(state)
    saved_log_probs.append(log_prob)
    state, reward, terminated, truncated, _ = env.step(action)
    rewards.append(reward)

    if terminated:
        break

# END_YOUR_CODE
```

4-(b) computeG

```
# 4-b
# Compute the discounted return for each time step.
# BEGIN_YOUR_CODE
for t in range(n_steps)[::-1]:
    disc_return_t = (returns[0] if len(returns) > 0 else 0)
    returns.appendleft(gamma * disc_return_t + rewards[t])
# END_YOUR_CODE
```

4-(c) computeLoss

```
# 4-c
# Maybe standardization of the returns is employed to make training more
stable
# eps = np.finfo(np.float32).eps.item()
# eps is the smallest representable float, which is
# added to the standard deviation of the returns to avoid numerical
instabilities
# BEGIN_YOUR_CODE
eps = np.finfo(np.float32).eps.item()
returns = torch.tensor(returns)
returns = (returns - returns.mean()) / (returns.std() + eps)
```

```
# Compute loss for each time step and sum them up.
policy_loss = []
for log_prob, R_t in zip(saved_log_probs, returns):
    policy_loss.append(-log_prob * R_t)

episode_loss = torch.cat(policy_loss).sum()
# We are actually using Gradient Accummulation here.
# So you need to divide the loss of an episode by the batch size.
loss = episode_loss / batch_size

# END_YOUR_CODE
```

6. 回答问题

6.1

状态空间过大、环境变化等

6.2

Model-free, Temporal Difference, off-Policy, Value-Based, online

[Optional]6.3

采样更多的轨迹，引入优势函数等