Project Report: Impact of compressed state representation for RL in UNO agents.

Group memebers: Ky Anh Vo, Adita Vashishth

1. Problem description

The problem we aim to tackle, which has been changed from project proposal, is to evaluate how a more compressed state presentation manages compares to a more robust state presentation through training time in reinforcement learning.

2. Environment setup, Agents setup, State presentation

Environment constraints:

- Player count for each game: 2 (due to RLCard constraints).
- Only 1 card at a time.
- No add card (+2, +4) stacking (e.g. if opponent plays +2 and agent has +4, agent must pick up 2 cards instead of stacking +4 and force opponent to pick up +6).

There are 2 different state representations that we test on, which we call Card and Strat.

Strat Representation (compressed)	Card Representation (robust)
For our hand: • 1 field each color (e.g. if we have 10 reds then the corresponding my-red = 10). • 1 field for suit: Number, Skip, Reverse, Draw2, Wild, and Wild-draw-4 (e.g. if we have 3 numbered cards then my-number = 3). 1 field for opponent card count For discarded deck, do the same as our hand. For target (top) card: • Top color (onehot, 1 field each color) • Top suit (onehot, 1 field each suit) • Top num (onehot, 1 field each number).	For our hand: there are 60 cards, so 60 fields, each field is the number of that card on hand (0, 1, or 2 cards). 1 field for opponent card count. For discarded deck: refer to our hand representation. For target (top) card: refer to Strat representation’s target card representation

3. Learning algorithm

For each of the state representations, we apply Deep Q-learning and Deep Monte Carlo learning, so we have 4 agents, named DeepQCardAgent, DeepQStratAgent, DeepMCCardAgent, and DeepMCStratAgent.

For each agent, after some n iterations, we train the agent's online network by keeping the tuple of states-target:

$$(S, \text{compute\_target}(S, S_{next}, \text{reward}, \text{action}, \text{done})),$$

where

$$\text{done} := \text{game ends after } (S, a) \; ? \; 1 : 0$$

and pass all the tuples into our model.

For DeepQ agents, we use the architecture of an online network for predicting plus a target network for stabilization in calculating Q value, and call compute_target as follow:

$$\text{compute\_target}(S, S_{next}, \text{reward}, \text{action}, \text{done}) := r + \text{gamma } (\max_{a'} Q'(S_{next}, a'), \text{done} = 0$$

$$\text{compute\_target}(S, S_{next}, \text{reward}, \text{action}, \text{done}) := (\text{win} \; ? \; 1 : 0), \text{done} = 1$$

where Q' is the target network.

After some N iterations (tunable) we do the operation

$$\text{target\_network.weights} := \text{online\_network.weights}.$$

For DeepMC agents, compute_target is as follow (n states recorded):

```
def compute_targets(S, S_next, r, a, done):
    target_lst := [0.] * state_count
    G = 0
    for i n...1:
        if self.dones[i]:
            G = 0
        G = self.rewards_list[i] + self.gamma * G
        target_lst[i] = G
    return target_lst
```

4. Hypothesis

On state compression performance: Strat state will perform better earlier in the training cycle compared to the more robust Card presentation, and it plateus earlier than Card presentation. The idea stems from the hypothesis that, earlier in the training cycle, Strat state modeling has higher chance of encountering similar states compared to Card state modeling. Consider a hand of red 3 and red 7, the other of red 4 and red 5. Strat state modeling can recognize that these are both 2 red hands, and able to move with that information, while the Card state modeling cannot see the similarity yet. However, later on Card presentation has a better view on the game in the sense that it has more specific card counting, creating an inherently more focused view on the game. It can map both colors and numbers (e.g. stacking yellows vs stacking 7) creating a more dynamic hand play.

5. Experiment

For the first 750,000 iterations, there are 4 of our training agents, 2 rule agents and 2 random agents. The 8 agents play with each other in a uniformly random manner.

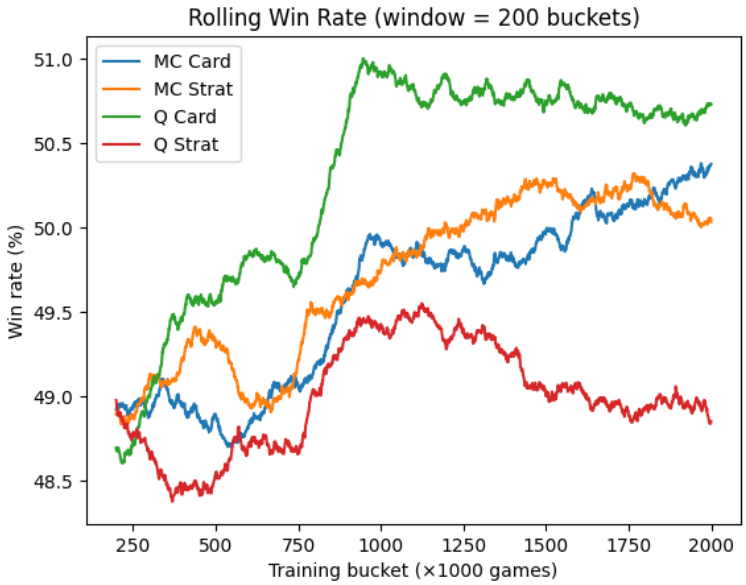
For the last 1,250,000 iterations, our 4 agents play in a self-play manner: 4 agents, uniform random distribution of matchups.

The intuition for such split is as the following: since we know that UNO is highly stochastic, we have elected to increase the window for epsilon decay of epsilon-greedy exploration so that the window of randomness decreases from p=95% to p=5% in 750,000 games, thus we set the baseline bots in there to produce semi-realistic scenarios for the agents to explore.

```
def train(total_games: int, training_agents: List[DeepUnoAgent]):
    rlcard_agents = []
    for _ in range(len(training_agents)):
        rlcard_agents.append(get_rule_based_agent())
        rlcard_agents.append(RandomAgent(61))
    all_agents = training_agents + rlcard_agents
    for game_idx in range(total_games):
        if game_idx < 7.5e5:
            play_games(all_agents, is_training=True)
        else:
            play_games(training_agents, is_training=True)
```

6. Result and Analysis

The experiment described above produces this graph of winrate for the 4 agents:



Note: self-play is defined as playing against other training agents.

Notice that for every model, there is a rise from bucket 750 to bucket 900. This can be attributed to the fact that, at this point, epsilon has now decreased to 5% and the games have turned from self-play and bot-play to only self-play.

In general, we can make the following observation:

- DQN Card agent seems to have plateaued early with relatively high winrate (around game 500-700).
- DQN Strat seems to struggle heavily in any scenario.
- Both Monte Carlos agents improves at the approximately equivalent rate until bucket 1500, where MC Strat plateues while MC Card seems to still be learning.

When considering only Monte Carlos agents, it seems like our state compression performance hypothesis is correct: in the first 500,000 games, MC Strat outperforms MC Card, though from that point to 1,750,000 games, the seemingly rises at the same rate, thought MC Strat plateues at 1,750,000 game and MC Strat keeps on moving up.

However, this hypothesis does not hold when we look at the major difference between the two DQN agents. DQN Strat seems to fail to learn anything past 500,000 games, but in the same timespan (0-500,000) DQN Card seems to learn much better, observable by the high sloping rise in winrate.

Thus our hypothesis for why the Strat state modeling fails for DQN: in Q learning, state compression introduces state aliasing that creates biased value estimates. Due to bootstrapping, we must thus reuse this Q value for a state with the same representation but completely different goals and win conditions. Thus the problem lies in the max operator of Q value, where the **max(a) Q(s, a)** that it tries to approach might be representing different states with different win condition. Thus we feed the incorrect state's Q value into the network training via compute_target, creating a positive self-reinforcing feedback loop where the Q values gets increasingly inaccurate over time. This

hypothesis can be applied to other bootstrapping algorithms due to them sharing the same logic of using the previously stored values to compute and train the newer values.

The reason that Monte Carlos does not suffer from this feedback loop is that it does not reuse its own incorrect answer to build its new answer upon, but it uses the gain/loss from the G values that it calculates over the course of one game.

7. Conclusion

In this project, we studied the impact of state compression on reinforcement learning agents in the highly stochastic, large state space game of UNO, comparing the effect of such state compression in both DQN and Deep Monte Carlos algorithms.

For Monte Carlos, state compression improves early training efficiency, but the outcome seems to not be relevant since it was only better in the first 500,000 games. After the preliminary heavy exploration cycle, both agents improve at the same rate until the compressed presentation inevetably reaches its ceiling.

In the other hand, for DQN, state compression consistently underperforms due to the mismatch of the max operator when handling state aliasing, creating a positive feedback loop. Future studies should try to implement state compression with other bootstrap algorithms (SARSA, A2C, A3C,...) to confirm or deny this hypothesis, and investigate different compressed state representations to avoid the current problem of mismatch goals for different states mapping to the same compressed state.