Build an AI to play Blackjack

1 Implement a Simplified Blackjack Game

Reinforcement learning learns an optimal policy through repeated interactions with an environment or simulator. Thus, you will first need to implement a simplified version of the game Blackjack. Your game will only have two players, one of which is the dealer.

- **Scoring a Hand**: A greedy algorithm will be used for scoring. Add up the score using 11 for aces, 10 for face cards, and the number for the numbered cards. If the score is $>$ 21 and the hand has 1 or more aces, convert the aces to 1 point, one at a time, to until the score is $<=$ 21.
- **Dealer**: The dealer has the magical ability to lookahead one card in the deck. The dealer will go first. The dealer will keep taking cards until the next card would cause them to bust.
- **Player**: The player is initially dealt two cards from the deck. They can choose to hit or stand. If they choose to hit, they are given another card. If they stand, the game is over.
- **Determining the winner**: If the player busts (goes over 21), the dealer automatically wins. If the dealer busts (but the player does not), the player wins. Otherwise, whomever has the highest-scoring hand wins. Ties go to the player.

2 Design Your Own Policy and Evaluate

A policy is a two-column table with one column for the states and another column for the optimal action for each state. (States may not be repeated.) The game will have 23 states: 1 – 21 for the current score of the agent's hand, win, or loss/bust. The agent may take two actions for states 1 – 21: hit or stand. For the win and loss/bust states, the action is automatically stand.

- Come up with your own policy table;
- Implement an player that plays the game using a given policy;
- Have the player play the dealer 1,000 times. Record the player's final hands and the number of wins and losses. Graph the distribution of scores.

3 Optimize the Policy with Q-Learning

Now that you have some intuition as to what a policy looks like and how to use it, you are going to use the q-learning algorithm to learn an optimal policy. There are several core aspects of the q-learning algorithm:

- **Rewards table**: in each state, the player has a set from which can choose an action. Taking that action will change the state. This process is stochastic so that player does not know what the resulting state will be. The rewards table records rewards for all possible outcomes of each action that can be taken in each state. It has four columns: current state, the action, the next state, and finally the reward earned if the next state is realized after taking the action in the current state.

  An easy way to represent a rewards table is to use a function. For our simplified blackjack game we will use:

$$r(s) = \begin{cases} -1000 & \text{if } s = \text{ loss} \\ 1000 & \text{if } s = \text{win} \\ s & \text{if } s \neq \text{win or loss.} \end{cases}$$

- q-**table**: The algorithm solves for a table of "quality" scores. The quality scores represent the expected future rewards resulting from an action. This differs from the rewards themselves which only give the the immediate reward and depend on the resulting state. The quality scores average over all possible future outcomes by trying different actions in each state and accumulating the results. The q-table has three columns: state, action, and q-value. Note that each state-action pair can only be in the table once. The q-values are initialized to zero at the start of the algorithm.

- **Action selection function**: The algorithm needs a function that chooses which action to take based on the current state and the current q-values. q-learning is effectively solving an explore-exploit problem. It needs to try all possible actions in each state but also evaluate the long-term result of choosing the best action. To implement this, write a function that chooses the current best action with probability $1 - \varepsilon$ and chooses a random action with probability ε:

  choose_action(current_state, q-value):

  t = np.random.uniform()

  if $t < \varepsilon$:

  action = randomly choose from all possible actions in the current state with uniform probability

  if $t > \varepsilon$:

  action = action with the highest q-value for the current state

To use use reinforcement learning to solve this problem we implement the q-learning algorithm:

optimize_policy():

initialize q-table with q-values of 0

for n games:

initialize game

while state $\neq$ win or loss:

determine the current state s

choose an action a to take using your choose_action function

take action a

determine the resulting state s' and the reward r

find the action a' associated with state s' with the highest q-value

update the q-value for state s with the following formula:

$$q[s,a] = (1-\alpha)q[s,a] + \alpha(r(s) + \lambda q[s',a'])$$

Finally, once we run the q-learning algorithm, we use the results to construct the optimal policy from the final q-table yielded by the algorithm:

construct_policy(q-table):

for each state s:

find action a with the largest q-value

add s, a to the policy table

Run the q-learning algorithm a few times. Use 0.1 for ε, λ, and α parameters. Compare the resulting policies. Try one of these policies using the approach in part 2.

4 Lab Report

Write up your analysis in a separate 2 – 3 page report. Your report should address:

1. Your hand crafted policy and the analysis.
2. The policy generated by the q-learning algorithm and the analysis.

Submit your lab report as a PDF and your code as a zip file in canvas.