**Reinforcement Learning**

Reinforcement learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward, where environment is typically formulated as a Markov decision process. Markov means action outcomes depend only on the current state. A Markov Decision Process is a discrete time stochastic control process. At each time step, the process is in some state s, and the decision maker may choose any action a that is available in state s. The process responds at the next time step by randomly moving into a new state s', and giving the decision maker a corresponding reward (s, s').

We use reinforcement learning to train a Q-learning agent to play Blackjack against with dealer machine. Dealer machine adapts a fixed strategy: hit only when the total value of its cards is less than 17, otherwise stand. We train our Q-learning agent with a lot of learning episodes to compute the Q-value map where the key is the state-action pair. Once the training completes, we could play against with dealer machine using the policy derived by choosing the optimal action with the largest Q value from Q-value map according to the current state.

Q-Learning Algorithm

Sample-based Q-value iteration:

In each episode, receive a sample (s, a, s’, r) and then compute new sample estimate Q(s, a):

Sample = R(s, a, s’) +

Update Q-value map using running average:

Q(s, a) (1 - ) Q(s, a) + ()[sample]

Use e-greedy action selection strategy to deal with exploration and exploitation:

Act randomly with small probability , act on current policy (derived from Q-value map) with large probability 1-. And lower over time.

Derive the optimal policy for all possible states:

For each state, compare Q-values for all possible actions and choose the best action with the largest q-value.

**Blackjack Q-learning definitions:**

A state is (dealerCard, playerTotal, hasUseableAce)

dealerCard is dealer's face-up card that player can see.

playerTotal is the total value of player's cards.

hasUseableAce is true iff playerHand's ace can be count as 11 and playerHand's total value is less or equal to 21.

A hand is (total, hasAce)

total is the total value of the hand's cards.

hasAce is true as long as the hand has an ace.

An action can be either “hit” or “stand”

States space:

Dealer card: a card from 1 to 11

Player’s total value of cards: from 2 to 21

HasUseableAce: True or false

So the total states space is 11 \* 20 \* 2 = 440

Pick cards method:

We randomly pick a card from 1 to 13, for the cards with values larger than 10, we count them as 10.

A counterMap is used to store how many times a particular state-action pair has been observed so that we can decrease over time according to the counter.

Select actions:

We use e-greedy action selection strategy.

Rewards computing:

We compute the reward by comparing the playerHand and dealerHand.

If player’s total value > 21, reward = -1

Else if dealer’s total value > 21, reward = 1

Else if player’s total value < dealer’s total value, reward = -1

Else if player’s total value == dealer’s total value, reward = 0

Else if player’s total value > dealer’s total value, reward = 1

**Blackjack Q-learning process:**

Initialize Q(s, a) map

Initialize counter map

Store all possible states

Repeat n learning times:

Select a state randomly

Initialize dealerFaceUpCard, dealerHand and playerHand

While True (For each round):

Select an action using e-greedy strategy

Formulate state-action pair (s, a)

Counter(s, a) + 1

If player hits:

Add a card to playerHand

If player does not bust:

Compute next state s’ by dealerCard and current playerHand

Compute max Q-value of the next state

Reward is 0 since we only know dealer card

Update Q(s, a) using running average, divided by counter(s, a)

Update state s to the next state s’

If player busts:

Terminal state, max Q-value is 0

Reward is -1

Update Q(s, a) using running average, divided by counter(s, a)

Break

If player stands:

Dealer play with a fixed strategy

Dealer continue to hit if current total value is less than 17

Terminal state, max Q-value is 0

Get reward by comparing dealerHand and playerHand

Update Q(s, a) using running average, divided by counter(s, a)

Break

Return Q-value map

**Results.**

We computed the average gain for our blackjack q-learning algorithm. For each round, if our q-learning agent wins, gain + 1. If it’s a tie, gain remains unchanged. If dealer wins, gain -1. For different number of training episodes, the average gain is the total gain / the total number of rounds. Here we set the total number of rounds as 100000.

X-axis is the number of training episodes. Start with 500 episodes, every time we multiply 2 as the next number of training episodes. We set the maximum number of training episodes as 4096000 episodes. Y-axis is the average gain of 100000 rounds.

The average gain becomes stable around -0.05 after 2048000 training episodes.

Below are the converged policy and the chart of the average gain:



