

## Learning to play blackjack

In this assignment, you will implement a reinforcement learning method that learns to play blackjack. Your program will be somewhat generic in that it will not have the rules of blackjack built-in — it will learn how to play as it goes along! This assignment is structured only in that you will need to compute the utilities of states. The support code provides the “performance element” that will actually play blackjack, making decisions on what action to take based upon the state, observed transition probabilities, and your learned utilities.

### Rules of blackjack

Blackjack is a card game in which each player at the table is playing against the dealer, and the dealer has a prescribed strategy. The basic objective is to have a “hand” with the higher value but not over 21.

- **cards** — a standard deck of playing cards is used, i.e., there are four *suits* (clubs, diamonds, spades, and hearts) and 13 different cards within each suit (2 through 10, jack, queen, king, and ace)

We will be using a *shoe* of 4 decks which will be reset after approximately 3 decks of cards have been played.

- **card values** — the numbered cards (2 through 10) count as their numerical value. The jack, queen, and king count as 10, and the ace may count as either 1 or 11.
- **hand value** — the value of a hand is the sum of the values of all cards in the hand. The values of the aces in a hand are such that they produce the highest value that is 21 or under (if possible). A hand where any ace is counted as 11 is called a *soft* hand.

*Blackjack* is a two-card hand where one card is an ace and the other card is any value 10 card.

The suits of the cards do not matter in blackjack.

### Procedure

There are some slight variations on the rules and procedure of blackjack. Below is the simplified procedure that we will use for this assignment — we will not be using “insurance” or “splitting” which are options in the standard casino game.

1. Each player places a bet on the hand.
2. The dealer deals two cards to each player, including him/herself. The players’ cards will be face-up. One of the dealer’s cards is face-up, but the other is face-down.
3. The dealer checks his/her face-down card. If the dealer has blackjack, then the dealer wins the bets with all players unless a player also has blackjack. If this happens, this is called a *push*, meaning that the player and dealer have tied, and the player keeps his/her bet.
4. If a player has blackjack (but the dealer does not), then that player wins immediately. The player is paid 1.5 times his/her bet.
5. Each of the remaining players, in turn, is given the opportunity to receive additional cards. The player must either:

- *hit* — the dealer gives the player an additional card (face up)
- *stand* — the player has finished his/her turn

A player can receive as many cards as he/she wants, but if the value of the player's hand exceeds 21, the player *busts* and loses the bet on this hand.

Before the player has received any additional cards, he/she may "double-down." This means that the player doubles his/her bet on the hand and will receive only one additional card. The disadvantage of doubling-down is that the player cannot receive any more cards; the advantage is that the player can use this option to increase the bet when conditions are favorable.

6. The dealer turns over his/her face-down card. The dealer then "hits" or "stands" according to the following policy:
  - If the value of the hand is less than 17, the dealer must "hit."
  - If the hand is a "soft 17," the dealer must "hit."
  - Otherwise, the dealer must "stand."

Most casinos force the dealer to hit a soft 17, but there is some variation in this regard.

If the dealer busts, then he/she loses the bets with all remaining players (i.e., those players that did not bust or have blackjack).

7. The dealer then settles the bets with the remaining players. If the dealer has a hand with a higher value than the player, then the player loses his/her bet. If the values are equal, then it is a "push" and the player keeps his/her bet. If the player has a hand with a higher value, then the dealer pays the player an amount equal to the player's bet.

## Basic Scheme representation and game states

Cards are represented by a list of two elements:

- the first element is either an integer between 2 and 10 or one of the symbols: `jack`, `queen`, `king`, and `ace`.
- the second element is one of the symbols: `diamonds`, `spades`, `clubs`, and `hearts`.

A hand is represented by a list of cards. This will be a list with a length of at least two. The first two elements will always be the cards that you were initially dealt, i.e., any additional cards you receive will be appended to the end of the list.

The "game-state" is a list where:

- the first element is the dealer's face-up card
- the second element is your hand

This, however, is not the "real" state that you will be using for reinforcement learning.

You will have to write a procedure (`transform-state game-state`) which takes a state as described above, and turns it into a "state number" which must be a non-negative integer. Here is a (probably not very good) example which turns the game-state into one of two states:

```
(define (transform-state game-state)
  (let ((my-hand (second game-state)))
    (if (>= (bj-value my-hand) 17)
        0
        1)))
```

The `bj-value` procedure will be available in the support code. It takes a “hand” and returns its value (a positive integer).

Here is a better example that turns the game-state into one of 23 states:

```
(define (transform-state game-state)
  (let* ((my-hand (second game-state))
        (value (bj-value my-hand)))
    (if (<= value 21)
        value
        22)))
```

You will need to declare how many states you are using for your reinforcement learning. For  $n$  states,  $O(n^2)$  storage is required, so you shouldn't have too many states.

The storage for this assignment (which will be part of the support code) will be stored in vectors which are indexed from 0 through  $(n - 1)$ . The second example above is slightly inefficient in that the smallest value that `bj-value` can return is 4, so states 0 through 3 will never be used.

## Reinforcement learning

I am still putting the finishing touches on the support code; it will be out before Saturday 11/22 but hopefully earlier. Rather than risk chaos by specifying details now that might change, I'll just outline how your reinforcement learning procedures will integrate into playing blackjack. You can expect another handout with the details, available online but also distributed in class on Monday.

You will have some choice as to which reinforcement learning technique you use. As a default, I would suggest temporal differencing.

Here is how the overall system will work. The “shoe” will be initialized with shuffled cards (and thereafter reset when necessary). Then hands are repeatedly played:

- deal cards to the (one) player and dealer
- repeat:
  - ask the player for an action
  - if “stand” then break (i.e. exit this loop)
  - deal a card to the player
  - if player busts, then give the learning element the state transition (with the new player hand) and reward (negative bet amount). Go on to next hand.
  - if “double-down” then break
  - give the learning element the state transition (with the new player hand) and a 0 reward (because the hand is not finished yet)
- play the dealer hand
- settle the bet
- give the learning element the state transition (with the new player hand) and a reward according to the outcome of the bet

## Support code and other details

Final details of the structure and procedures you must write as well as a few written questions will be out before the weekend. Check the web page for details.