

Learning to play blackjack

In this assignment, you will implement an active reinforcement learner for playing 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! The support code will run the blackjack game and call your code to decide on what actions to take and for it to learn utilities of the states.

A. Rules of blackjack, representations, and playing blackjack

Blackjack is the most popular casino card game. A number of players may be seated together at a blackjack table, but players are each playing against the dealer individually, not against each other. The basic objective is to have a “hand” with a value higher than the dealer’s, but not over 21. The players can choose how they play their hands, but the dealer’s strategy for playing his or her hand is fixed.

A.1 Terminology

- **cards** — a standard deck of playing cards is used, i.e., four suits (clubs, diamonds, spades, and hearts) and 13 different cards within each suit (the numbers 2 through 10, jack, queen, king, and ace)
We will be using a *shoe* of 4 decks. This means that four complete decks of cards are shuffled together and placed in the “shoe” from which cards are drawn. The shoe will be reset after approximately 3 decks of cards have been played. (The shoe is reset in between hands, not during a hand.)
- **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. The suits of the cards do not matter in blackjack.
- **blackjack** is a two-card hand where one card is an ace and the other card is any value 10 card.

A.2 Rules of play

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- or 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 or 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 or her bet.
5. Each of the remaining players, in turn, is given the opportunity to receive additional cards. The player must either:

- *hit* — the player receives one additional card (face up). A player can receive as many cards as he or she wants, but if the value of the player’s hand exceeds 21, the player *busts* and loses the bet on this hand.
 - *stand* — the player does not want any additional cards
 - *double-down* — before the player has received any additional cards, he or she may double-down. This means that the player doubles his or 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 (beyond the one additional card); the advantage is that the player can use this option to increase the bet when conditions are favorable.
6. The dealer turns over his or 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.

If the dealer busts, then he or 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 or her bet. If the values are equal, then it is a “push” and the player keeps his or 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.

In this assignment, there will only be one player; if the player busts, then the dealer’s hand will not be played.

A.3 Scheme representation

Note that the support code provides procedures (described in the next subsection) to calculate the value of a hand.

- A card is 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.
 - A player’s hand will always be a list with at least two elements. 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.
 - A dealer’s hand will consist of a list of one card (the face-up card) before the dealer has played his or her hand. After the dealer’s hand is played, it will be a list of all the dealer’s cards; the first element will still be the dealer’s original face-up card.

A.4 Support code for evaluating hands

- `(bj-value h)` — returns the value of the hand `h`.
- `(soft-hand? h)` — returns `#t` if the hand `h` is a soft hand.
- `(blackjack? h)` — returns `#t` if the hand `h` is a blackjack.
- `(card-string c)` — returns a string containing an abbreviated version of the card (useful for printing debugging information).
- `(display-hand h)` — prints an abbreviated version of all cards in the hand `h`.

A.5 Players and strategies

A *player* is a list of three things, in order:

1. A (double-quoted) string containing the name of the player. You can use any name except "dealer".
2. A *strategy procedure* of the form `(lambda (fs actions) ...)` where `fs` is a reinforcement learning state, and `actions` is a list of actions that your player may take at that point in the game.
3. A *learning procedure* of the form `(lambda (fs a ts) ...)`. This procedure will be called on when there is a state transition from state `fs` after taking action `a` to state `ts` (where `fs` and `ts` are reinforcement learning states).

See the `a7example.scm` file for an example.

A.6 Playing blackjack

There are two procedures for playing blackjack with your player:

```
(play-hand player)
(play-match N player)
```

The first plays a single hand. The second may be used to play a fixed number of hands (when `N` is a positive integer) or to play hands repeatedly until some condition is met (when `N` is a procedure that takes 0 arguments).

The initial bet on each hand is 1. If the player doubles down, then the bet is increased to 2. The `play-match` procedure returns a list of two numbers: the first is your player's net winnings, and the second is the total amount your player bet. Note that for a fixed number of hands, the amount bet will vary if your player doubles down.

Note that you must have initialized the tables (see Section D.) and have defined the `calc-x-state` procedures (Problem 1) in order to play blackjack — even if your player does not use them (as is the case for the random player). Again, you can use the procedures in `a7example.scm` to get started.

The operation of `play-hand` and `play-match` can be controlled by the following variables:

- `print-narration` — when `#t` (its default value) a narration of the play is printed to the screen. You can disable printing parts of this information with the following variables (all of whose default value is `#t`): `print-player-play`, `print-dealer-play`, and `print-bet-settlement`.

When `print-narration` is set to `#f`, no narration is printed to the screen, with one exception, controlled by the following variable:

- `print-match-progress` — when set to, for example, 10 (its default value), a message will be printed every 10 hands. If you don't want this message to be printed, you can set it to `#f`.

- `print-learning` — when `#t` (its default value), a message will be printed before each call to your learning procedure.
- `enable-table-updates` — when `#t` (its default value), the support code records all state transitions and rewards in order to build up data for estimating state transition probabilities and average rewards. Disabling table updates will preserve the current transition probabilities and average rewards.

While testing, you may find that you want to run your code on the same sequence of cards/hands. In order to do this, I have provided procedures that let you manipulate the state of the random number generator.

- `(save-random-state)` — makes and saves a copy of the random number generator state
- `(restore-random-state)` — restores the random number generator state to that from the last call to `save-random-state`. The sequence of random numbers (and therefore sequence of cards) will be the same as following the last call to `save-random-state`.

You will notice that if the dealer or the player has blackjack, there is no update to the tables, nor is the player's learning function called. This is because there is no state transition in these situations.

B. Reinforcement learning states

A reinforcement learning method will calculate utility values for states. You will need to write two procedures to transform the "game state" (i.e., the dealer's and player's cards) into a "reinforcement learning state" (represented by a nonnegative integer). You will have to decide how to do this transformation. I will provide a simple example in the `a7example.scm` file.

The support code will provide procedures for storing utility values for reinforcement learning states as well as for keeping track of transition probabilities between reinforcement learning states.

Problem 1 (16 points) Write the following procedures: `calc-initial-state` and `calc-new-state` that transform the game state into a reinforcement learning state, and `init-tables` that initializes the tables for transition probabilities, rewards, and utility values.

Here are details on each of these procedures:

- `(calc-initial-state player-hand dealer-hand)`
The argument `player-hand` will consist of a list of two cards that the player was initially dealt; `dealer-hand` will consist of a list of one card, the dealer's face-up card. This procedure must return a valid reinforcement learning state (a nonnegative integer between zero, inclusive, and the number of states declared to `create-tables`, exclusive).
- `(calc-new-state previous-rl-state action terminal? reward player-hand dealer-hand)`

The arguments are as follows.

- `previous-rl-state` will be a nonnegative integer (returned by either your `calc-initial-state` procedure or by the previous call to your `calc-new-state` procedure).
- `action` will be one of the symbols `hit`, `stand`, or `double-down`
- `terminal?` will be `#t` or `#f` depending on whether the action resulted in a terminal state or not. The `stand` and `double-down` actions always result in a terminal state. The `hit` action may or may not result in a terminal state (depending on whether you bust or not).

- reward is the reward for the new state. For nonterminal states, the reward will always be zero, but for terminal states, this will be the amount that you win or lose. (Note that this might be 0 if there is a "push.")
- player-hand will be the player's hand *after* the action is taken. For the hit and double-down actions, there will be a new card added to the end of this list. For the stand action, there will be no change.
- dealer-hand will be the dealer's hand after the action is taken. For the stand and double-down action, the dealer's hand will have been played, so this argument will contain all the dealer's cards. For the hit action, the dealer's hand will consist of only the face-up card, even if the player has busted.

It must return a valid reinforcement learning state.

- (init-tables)

This procedure must create and initialize the tables in the support code that store utility values, rewards, and transition probabilities. See the Section D. for details.

C. Reinforcement learning

If the utility values are known, then the best action is the one that maximizes the expected utility.

Problem 2 (16 points) Write the strategy procedure (`basic-rl-player fs-num actions`) that takes a nonnegative integer `fs-num` and a list `actions` that are permissible. It should return the action that maximizes expected utility. Essentially, your procedure should implement the following equation:

$$a = \operatorname{argmax}_{a_i \in A} \sum_{s'} T(s, a_i, s') U(s')$$

Note: your `basic-rl-strategy` procedure should be able to work with any valid solution for Problem 1 (including the one I will provide in the `a7example.scm` file).

However, we are interested in learning the utility values as our agent plays the game. To do this, you will implement temporal differencing. There are two parts:

Problem 3 (12 points) Write the procedure (`create-exploring-rl-player R+ Ne`) that returns a strategy procedure that incorporates the simple exploration function described in our text (page 774) where `Ne` and `R+` are parameters in the exploration function.

Problem 4 (12 points) Write the procedure (`create-td-learning alpha`) that returns a learning procedure that implements temporal differencing. This learning procedure will be called for each state transition, so it should update the utilities according to the temporal differencing update equation:

$$U(s) \leftarrow U(s) + \alpha(R(s) + U(s') - U(s))$$

The `alpha` argument will be a procedure of the form (`alpha n`) where `n` is the number of times that there has been a transition from state `s`.

Note that this is equation 21.3 from our text, but I have omitted the discount factor γ and changed the notation from $U^\pi(s)$ to $U(s)$ since we do not have a fixed policy π here.

D. Utilities, transition probabilities, and rewards

The support code will keep track of all observed state transitions and rewards in order to build up a model of blackjack for your states. It will also provide storage for the utility values for nonterminal states.

D.1 Tables

In Problem 1, you need to write the `init-tables` procedure. At least initially, all this procedure has to do is call the following procedure in the support code:

- `(create-tables num-states)`

The argument `num-states` is the number of states you will use; the state numbers are 0 through `(num-states - 1)`. The optional argument `utility-init-proc` must be a procedure that takes zero arguments. This procedure will be called to initialize the utility value for every nonterminal state (as indicated by your `terminal-state?` procedure). If not given, the utilities will be initialized to zero.

Each time this procedure is called, it will create new, empty tables; any information in the old tables will be lost.

There are a number of other procedures to view and save all the tables:

- `(num-states)` — returns the number of reinforcement learning states.
- `(save-tables fname)` — given a double-quoted string argument, it will print definitions of the transition, utility, and rewards tables to a file of that name.
- `(print-rl)` — prints all the tables by calling `(print-transitions)`, `(print-rewards)`, and `(print-utilities)`. All these procedures are described in the following subsections.

There are several variables that control how the tables are printed to the screen:

- `print-line-width` — default value is 80
- `transition-decimal-places` — default value is 3
- `utility-decimal-places` — default value is 3
- `reward-decimal-places` — default value is 3

One more (somewhat technical) note: the support code counts the number of “action transitions,” i.e., the number of times a given action is tried from each state. If you ran the random player lots of times to learn the transition probabilities and rewards, all its actions are recorded in this manner. However, when you first start a temporal differencing program, you want it to start exploring the states without any knowledge of what happened previously. This is necessary for your exploration function to work. To reset the action transition counts, call the procedure:

- `(reset-action-transitions)` — all action transition counts (as returned by the procedure `get-action-transitions`) will be zeroed by a call to this procedure.

Utilities

For nonterminal states, your reinforcement learning algorithm will compute the utility values. For terminal states, the utility will be the average reward received in that state.

The following procedures are available in the support code to access the utilities:

- `(print-utilities)` — prints a table of the utility values to the screen
- `(get-utility-element state-num)` — returns the utility of the reinforcement learning state `state-num`
- `(set-utility-element state-num u)` — changes the utility value for the state `state-num` to `u`.

Transitions

The transition probabilities describe the probability of going from state to state when a given action is executed. In our text's notation, these are the $T(s, a, s')$. They are estimated by simply dividing the number of state transitions from s to s' when action a is executed by the total number of times action a was taken from state s .

The following procedures are available in the support code to access the transition probabilities:

- `(print-transitions)` — prints the transition table to the screen
- `(get-transition-probability fs-num action ts-num)` — returns the probability of transitioning from state `fs-num` to state `ts-num` under `action`. `fs-num` and `ts-num` are reinforcement learning states (nonnegative integers), and `action` is one of the symbols `hit`, `stand`, or `double-down`.
- `(get-transition-alist fs-num action)` — returns an association list of all transitions from state `fs-num` under `action`. The result is a list of the form:

```
((ts-num1 tprob1) (ts-num2 tprob2) ...)
```

where the `ts-num`'s are states for which there was an observed transition, and the `tprob`'s are the fraction of times (i.e., probability) of that transition.

- `(get-action-transitions fs-num action)` — returns the number of times that `action` has been taken from state `fs-num`

Rewards

The support code also keeps a running average of what rewards are received in every state. Note that, in general, the actual reward is random because it depends the value of the dealer's hand. The following procedures are available in the support code to access the rewards:

- `(print-rewards)` — prints a table of the reward values to the screen
- `(get-reward state-num)` — returns the average reward received in the reinforcement learning state `statenum`

E. Conclusion

Problem 5 (84 points; written and electronic) I will ask you to learning a blackjack player with different parameters for an exploration function and for temporal differencing. You will turn in your final player and write up details of your implementation. See the assignment web page for details on this problem.