Methods:

Both of us worked extensively, in collaboration with one another, on all aspects of the project but we each took lead on a different aspect of the project.

The first step we took was to implement a Blackjack game for our AI to use. James was the primary author for these methods, and Andrew was the secondary author. We found the rules for blackjack on a website called wizardofodds.com. There are many variations of blackjack, with the rules changing depending on the casino the game is played in. For our implementation we used the following set of rules.

* The player may hit, stand, double down or split
  + Hit means the hand gets another card
  + Stand means the player keeps their current hand and passes the turn
  + Double down means the player doubles their bet and hits only once
  + Split means the player splits the two cards in their initial hand and adds a new card to each of the split cards so they have 2 hands. They must match their initial bet on their new hand. A player can only split if the two cards have the same rank (ex [J, J] becomes [J, X] and [J, X])
* A player may split as many times as they are able.
* The dealer must hit if the value of their hand is less than 17
* The dealer must stand on 18 and above
* The dealer must stand on 17 unless it is a soft 17
  + A soft 17 is defined as a hand with a value of 17 that contains an ace whose value is 11.
* Blackjacks pay the player 3:2
  + Blackjacks are defined as a hand with 2 cards that total to 21
* Wins pay 1:1
* Losses result in the player losing their bet
* Draws result in a push, and the player gets their bet back

Our implementation of blackjack uses a dealer, and a single player which was used as the agent when training the games Q table. We also implemented a card class to keep track of the details of any given card, a deck class that contains the remaining cards in the deck, and a hand class that keeps track of the cards in the hand of a player or dealer.

The dealer class is the main engine of the game. It keeps track of the dealer’s hand, as well as the player that is playing against the house, and the deck of cards being used by the game. It is responsible for dealing new hands from the deck to the player and dealer, as well as dealing cards from the deck to the player when the player hits. The dealer plays his hand only after the player has finished, and only has the options of hitting or standing. It follows a strict set of rules, outlined above, when playing its hand.

The player class contains the player’s bankroll, a list of hands as the player can have more than one, as well as a list of bets associated with each of their hands. The player’s possible moves are to hit, stand, double down, and split. We opted to use a separate function to determine the valid moves of a hand, but in hindsight, it would have been better to give that responsibility to the player.

The deck class maintains a list of cards that remain in the deck. It is responsible for randomizing the deck when it is created and returning the top card of the deck when the dealer draws a card. It has the capability of being made up of any number of decks. For our experiment the deck is composed of six regular decks of cards.

The card class keeps track of the rank and value of a card. It also flags whether the card is an ace since an ace can have a value of 1 or 11.

The hand class maintains a list of cards in a hand. It is responsible for adding and removing cards from the hand as the player and dealer make moves. It also determines if a hands value is a blackjack, or if the hand has busted. The hand class is one of the workhorses of the game.

The blackjack game itself uses these classes to play a game of blackjack. It is composed of a player and a dealer and is responsible for determining the status of the game by comparing the player’s hands with the dealer’s hands to determine if the player has won, lost, or drawn with the dealer.

The second step we took was to write the AI to train the game. Andrew was the primary author for this portion, and James was the secondary author. For this step, our primary resource was assignment A5 and Chuck Andersons course notes on the topic of Q tables. We choose to use a Q table to store the states of the game and the outcomes of various moves. After training this Q table was then used to play 1000 hands of blackjack so the results could be analyzed.

In order to train the Q table that the AI would use to make choices we needed a function to determine the valid moves of hand, as well as a function to make a move. Our validMoves function takes a hand and determines if the player can hit, stand, double down, or split the cards based on the rules of blackjack. The makeMove function calls the methods in the player class to modify the selected hand based on the players selected move. Sometimes the first iteration of an implementation is not always the best, and if we were to rewrite the implementation we would give the player the responsibility of determining a valid move and the blackjack game the responsibility of making a selected move.

We also needed a way of representing the state of the game. To do this we chose to use a tuple of the value of the players hand, the dealers face up card, and the chosen move for the hand. We chose to use the value of the players hand because a hand of [6, 3] has the same value as [7, 2] and we wanted the AI to recognize this and choose the same move. This caused a bit of a tradeoff however, since an Ace paired with any other could should be viewed differently than another pair of cards that totals to the same value. We realized this later, when we were finishing testing and could not devise a simple fix to the predicament. The advantage to this however, is that more hands mapped to the same map state and trained the Q table faster.

We also implemented a function epsilonGreedy to slowly shift away from random moves towards using the Q table to choose its moves. This is important, because it allows the training function to start by randomly selecting its moves when the table is not populated. After the table is populated however, we want the trainer to select the move it deems best from the table and continuously test that move. If it is a good move it should retain a higher value than the other moves in the table otherwise it will slowly be replaced by a better move. In this way the Q function trains more efficiently than just randomly selecting its moves during the entire training session.

Finally we implemented the function to train the Q table. It takes number of repetitions, learning rate, epsilon decay rate, valid moves function, and a make moves function. The learning rate determines how much the Q value is affected of the states leading to the final state of a series of moves. The epsilon decay rate determines how quickly the trainer shifts from random moves to “intelligent” moves.

For each repetition the trainer creates a new game of blackjack which it plays until the deck, composed of six decks, has only one deck of cards remaining. This is our buffer deck. For each hand dealt, the player and dealer play out their respective hands and the game compares the outcome and updates the Q table accordingly. Draws do not affect the Q table values for the hands, while wins increase their state value and losses decrease their state value. This means for any given state, after training the move with the highest value in the Q table will be the move that training has determined to be the best move from this state.

https://wizardofodds.com/games/blackjack/basics/