Using Reinforcement Learning to Create Blackjack AI

David Sammel, CS4701

1. Introduction

The game of blackjack is one of the most popular and well-studied gambling games in the world. It has piqued the interest of gamblers and scholars for its basis complex probability. As a result, many strategies have been formulated to try to maximize the chance a player can win the game. The work in this project uses Artificial Intelligence, specifically Machine Learning, to learn a strategy to play hands of blackjack, and win them at a fair rate.

The premise of the game is relatively simple, have the sum of your cards be greater than the dealer's without that sum going over twenty one. Each card is worth the number on it, face cards are all worth ten, and aces are worth either eleven or one, at the player's choice. Each player and the dealer are dealt two cards, one face up and one face down. If any player has twenty one in their first two cards (an ace and a card valued at ten), he or she has blackjack and automatically wins. If the dealer has blackjack, all of the players automatically lose. Otherwise, the players can choose to hit, where they are given another card face up, or stand where they finish their turn. If any player or the dealer goes over twenty one, they bust, and automatically lose. Each player takes a turn, and then the dealer plays. Most commonly, the dealer will follow a strict strategy of hitting until his score is greater than or equal to seventeen. After the dealer finishes, each player's score is compared to the dealer's. If a player's sum is greater than the dealer's the player wins and if not, the dealer wins.

The main question a player must ask while playing blackjack is given the state of the game, is it more beneficial to hit or stand? This is the focus of much of the research into the game. The state of the game can be defined in a few ways. The most basic assessment of state is done in the

case of the dealer, where only the cards currently in the dealer's hand are considered as state. This definition of state makes sense for the dealer as he must follow a strict rule and his play style does not depend on any other players. The most common strategy for a player is to consider the state as the player's hand and the up-facing card of the dealer. The player's goal is to maximize the chance that his total will be higher than the dealer's. By considering the visible card the dealer has been dealt as well as his own, a player can make a more informed decision between hitting and standing.

A third strategy considers even more state of the game. When employing a card counting strategy, a player keeps track of not only his cards and the dealer's face up card, but the cards that have been seen in the current deck. Knowing the values of the cards that have already been dealt can help the player predict which cards are left in the deck to be dealt. If the deck contains more high cards than low cards, the player knows not to hit when he has a hand in risk of busting. This state of the deck also adversely affects the dealer because the dealer is forced to hit until his hand sums to at least seventeen. By the same analysis of the player's situation, the dealer is more likely to bust when the deck contains primarily high valued cards. The inverse is also true; a deck primarily of low cards negatively affects the player, and helps the dealer. When the dealer is at risk of busting, but not at seventeen yet, he is less likely to bust with low cards in the deck. The player is forced to hit more often in order to beat the dealer who will likely have a higher sum than usual. Thus information about the state of the deck can also be useful in creating a blackjack strategy.

The basic premise of counting cards is to assign point values to each card, and keep a running point total, or count, of the deck to determine its current state. Generally, low cards (twos, threes, fours, fives, and sixes), are assigned a point value of positive one, while high cards (tens, jacks,

queens, kings, and aces) are assigned a point value of negative one. Middle cards (sevens, eights, and nines) do not affect the deck significantly, so they are assigned a value of zero. Whenever a card is shown to the table, the player adds the card's point value to the running total. If the running total is positive, the deck is said to be high, since the remaining cards in the deck are higher than average. If the count is below zero, the deck is said to be low, and a card drawn from the deck is more likely to be low-valued than normal. Thus, by counting cards, the player can adapt his strategy to the current state of the deck.

2. Assumptions and Simplifications

The code of this project makes a few assumptions and simplifications to the game of blackjack. The assumptions are made in order to define certain aspects of the game more concretely. The simplifications are to create a simulation that can most accurately assess the problem I am trying to solve: make AI to play blackjack with a reasonably high winning percentage.

My first assumption is that the deck is reshuffled once half of the beginning cards have been dealt. Shuffling can occur at any time at a casino but for the framework this agent will be learning in, the time when the deck is shuffled is fixed.

My first simplification is to remove the aspects of splitting a hand and doubling down on a hand during play. Both are methods to increase payout while betting on the game, but have little or no effect on the chances of winning a hand. Second, I will assume all aces are valued at eleven, unless an ace is drawn that causes a hand to bust. In that case, it is valued at one. This simplification was used because the model will be learning whether or not to hit based on the current sum of the cards, not based on the breakdown of each card in the hand. For example, a

hand consisting of a queen and a six is considered equal to a hand containing an ace and a five (both totaling sixteen). However, if either of these hands hit and received an ace, their new total would be seventeen; because the ace recorded as worth eleven would have caused a bust.

3. Project Setup

The goal of this project is to create an agent that plays the game of blackjack, and learns when to hit and when to stand. First, there is an agent that takes into account its own hand's sum and the face up card of the dealer. Second, there will be an agent that will take into account the count of the deck as well.

The method to train the agents that has been used is to keep track of a probability that the agent will hit for every state of the game. When an agent plays its turn, it hits with probability p, and stands with probability $(1-p)$. The agent continues to play its turn until it either stands, or busts. At that time, the agent will record what the state was before the last hit, and then wait for the dealer to play. Once the dealer has finished its turn, the agent updates its hit probabilities. If the agent busted, the probability that the agent will hit in the recorded state will be lowered, and if the agent lost to dealer by having a lower point total, the probability the agent will hit in the recorded state is increased. If the agent beats the dealer, all probabilities remain unchanged.

Once the agents have learned rules to play by, other agents that play by the learned model can then be run to compare the model's win percentage against other players. One of the other players will be employing the dealer's strategy, and the other will be randomly guessing to hit of stand with equal probability (the learning agents' original models). With data from all of the models, evaluations can be made of the learned model.

4. Explanation of Code

In this section, I will explain the various modules, classes, and functions in my code. All code has been written in Python.

4.1 Constants.py

This module contains all the constants for the code. They are:

- NUMDECKS: the number of 52 card decks to comprise the game's deck.
- NUMHANDS: the number of hands to be played
- PRINTSTEPS: the number of hands between print statements
- FACEUP: A constant that should not be changed, increases code readability
- FACEDOWN: A constant that should not be changed, increases code readability
- HIGHTHRESH: The threshold for determining if the deck is high. If the true count is greater than HIGHTHRESH, then the deck is considered high
- LOWTHRESH: The threshold for determining if the deck is low. If the true count is greater than LOWTHRESH, then the deck is considered low

4.2 Deck.py

This module contains the code for the deck of cards to be used in the blackjack simulation. The class is initialized accepting one variable: the number of decks to be shuffled together to form the playing deck. If the initializer is called with an argument of four, the game will be played with 208 cards (52 cards/deck times four decks). The class initializer initializes four fields. The first field, *numberofdecks*, is a field that keeps track of the number of fifty-two card decks contained within one blackjack deck. The second field, *count*, is the current count of

the cards that have been already dealt and seen by the table. The third field, *truecount*, is the current count of the deck divided by the number of decks being used in the game. This value is more pertinent to card counters because when there are more decks, the effect of removing one card on the state of the remaining cards is reduced. This field is referenced often by the players to determine the state of the game. The final field, *deck*, is a list of integers representing the cards remaining in the deck.

The next method is called *shuffle*. This method creates a new list of cards and is called on initialization or when the list of cards remaining in the deck is half of the deck's original size. For each deck in *numberofdecks* the method creates four of each numbered card and ace (since there are four suits), and sixteen cards valued at ten. Suit can be ignored because it serves no purpose in blackjack. The list is then shuffled using a function from the imported *random* module, the *count* is reset to zero, and the *deck* field is set to be the new list.

The next method, *countcard*, handles updating the count of the deck. Given a card, the *count* field is either incremented, decremented or left unchanged by the rules described for counting above. The *truecount* is also updated to reflect the new card drawn as well.

The *dealcard* method handles dealing a card from the deck. Its only input is a Boolean describing if the card is to be dealt face up or face down (face up corresponding to True). If the card is face up, it is counted as it is dealt, if not, then it is counted when it is revealed at the end of the hand. The method takes the first element of the *deck* list, counts it if necessary, removes it from the *deck*, and returns it.

Outside of the class definition, the variable *gamedeck* is created. This is a Deck object initialized with the constant *NUMDECKS* decks.

4.3 Players.py

This module contains the code for the AI that is playing the blackjack, both in the learning phase and in the post-learning playing phase. This module contains classes for each of the play styles. A few of the classes contain functions that perform similar tasks done in similar ways, so they will only be explained once. Also, some of the classes are mostly identical, so they will be explained together.

4.3.1 Dealer

This class contains the code for the Dealer's play style. The player is initialized with fields for its hand (*hand*), its facedown card(*hiddencard*, for counting purposes), its number of wins (*numwins*, for statistics), if the player has busted on the current, and if the player has blackjack (*bust* and *blackjack*, respectively, for determining wins and losses).

The *play* method is the code that handles the playing of a hand. First, if the player hand is twenty one before hitting, the *blackjack* field is set to True. Next, while the dealer's hand is less than or equal to sixteen, the player will hit. This includes getting a new card from the deck, and adding the card's value to the *hand* field. Additionally, there is a check to see if the card drawn is an ace and causes the player to bust. As mentioned above, in this case the ace is treated as a one instead of an eleven. Finally, the hand is checked to see if the player has a bust. Once the player stands, the function returns.

The *receivecard* method handles receiving the first two cards when the hand is dealt. It is called in the main game playing function in other modules. It takes in a card and adds the card's value to the *hand*. If the card is the first card dealt in the hand, *hiddencard* is set to that card. Finally, if two aces have been dealt, the *hand* field will be reduced from twenty two to twelve.

The *win* and *lose* methods handle resetting the player's fields to be ready for another hand to be played. In *win*, the field *numwins* is incremented to keep the game's statistics up to date.

The *printstats* method prints out the current number of wins and win percentage for this play style when called.

4.3.2 Learner/CountingLearner

These classes are the agents that learn rules to play blackjack. The Learner class learns taking into account its hand and the dealer's face up card, while the CountingLearner class also takes into account the count of the deck. The initializers of these classes contain the same fields as the Dealer class, along with some extras. The *dealer* field keeps track of the dealer player in the game. *dealershow* contains the face up card of the dealer stored in *dealer*. The field *lasthit* is to keep track of the player's hand total before it took its last hit. In the Learner class, there is a *hit* field which is a two-dimensional list that contains the probabilities the agent will hit based on its hand and the dealer's face up card. The CountingLearner class has the fields, *midhit*, *highhit*, and *lowhit* which are initialized identically to *hit*. These fields store the probability the agent will hit (given its hand and the dealer's face up card) when the deck is neutral, high, and low, respectively.

Like the Dealer class, the *play* methods in these classes also handle playing a single hand of blackjack. First, the *dealershow* field is updated and the player's hand is checked for a blackjack. Then, the agent will hit with the probability stored in its probability lists. The CountingLearner class will first determine the list to perform the lookup in based on the count of the deck. If the agent hits, it will update its *lasthit* field, get a new card, add that card's total to

hand, and check for a bust. If it hasn't busted, it will continue to hit until it stands (1 – the stored probability of hitting in the given state), or it busts.

The next method that is any different from above is *lose*. This method updates the probability of hitting for the state the agent last hit in if the agent lost the hand. First, there is a check to see if the agent lost due to a dealer blackjack. Since a loss in this fashion isn't an effect of the agent's play style, the agent will not perform the update in this case. Otherwise, if the player busted, the probability of hitting in the proper list is reduced and if the player was beaten without busting, the probability of hitting is increased. Then, the fields are reset for the next hand.

4.3.3 Learned/CountingLearned

These classes are for agents that play using the learned rule, but don't update the rule. The only difference between these and the previous two classes is that they load a rule from .csv files upon initializations, and don't update the rule on a loss. These classes are made for statistical and demonstration purposes after the learning phase is over.

4.3.4 Counter

This class plays with the same strategy as the Dealer class, but records statistics on win percentages. It contains extra fields to keep track of the number of hands played when the deck is high or low, and the number wins the agent has in those circumstances. The *win*, *lose*, and *printstats* fields are updated intuitively to reflect this.

4.3.5 DumbPlayer

This class plays with the strategy the learning agents begin with. That is, it randomly hits or stands with equal probability regardless of state. This too is for statistical purposes.

4.4 BlackJack.py/Learn.py

Both of these modules handle playing the game of blackjack. Learner.py is for the learning stage of the game, and BlackJack.py is for the playing stage to assess the learned models. Both are very similar so they will be explained together.

4.4.1 Game

This is the class for the game object. It initializes the *dealer* field as a Dealer object, and the list of players in the *players* field. In Learn.py, the *players* list contains a Learner object and a CountingLearner object. In BlackJack.py, the *players* list contains Dealer, Counter, Learned, CountingLearned, and DumbPlayer objects.

The *dealhand* method iterates over the *players* field twice, first giving each player a face down card, and then each player another card, face up.

The *playhand* method plays a single hand for all of the players. First, the *play* method for each player is called, followed by the *dealer*'s *play* method. Once the *dealer*'s *hiddencard* is counted, each player is iterated over again. Each player's *hiddencard* is counted, and then each win and lose condition is checked. If the player won, its *win* method is called. Otherwise, its *lose* method is called. Once every player's is handled, the *lose* method for the *dealer* is called to reset its hand.

4.4.2 __main__

In the main function for both modules, a Game object is made. Then, NUMHANDS hands are dealt, and played. After each hand, the deck is checked to see if it should be reshuffled. If half of the cards in the original *gamedeck* have been used, the deck is reshuffled.

Once NUMHANDS hands have been played, the player's stats are printed using their *printstats* methods. In Learner.py, the learned rules are saved to .csv files for viewing, or loading by the Learned or CountingLearned classes.

5. Assessment

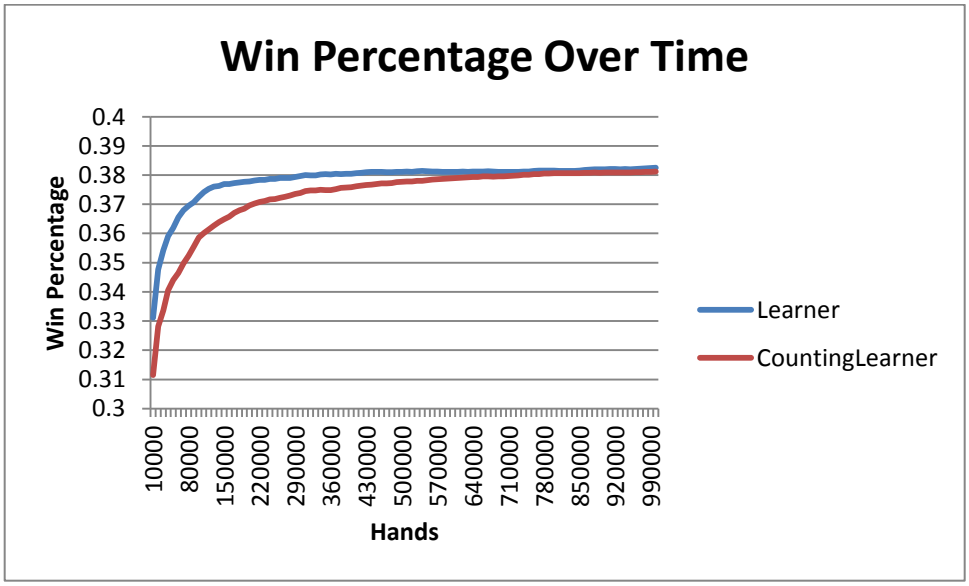
After training the learning models with Learn.py with 4 decks and 1,000,000 hands, the accuracy rates over another 1,000,000 hands running BlackJack.py are as follows:

Style	Dealer	Count (high)	Count (low)	Dumb	Learner	CountingLearner
Win pct.	41.4%	41.0%	42.1%	29.9%	38.3%	38.7%

The results show allow for a few conclusions about these agents. Most significantly, the learning agents show clear improvement over their "dumb" counterparts. This provides evidence that the algorithm in place to make the agents learn a rule to play blackjack by is effective. While the neither learning agent plays as well as the player matching the dealer's strategy, their winning percentages are much closer to that of the player matching the dealer than they are from that of the random guesser. Also, to reach almost a 40% win rate means the agents are playing rather effectively. The odds of winning a hand of blackjack are skewed in favor of the dealer, since the dealer wins all tiebreakers.

Also worth noting is the effect of counting cards on the agent's performance. Although only the agent counting cards wins 0.4% more games, over 1,000,000 hands, this is a statistically significant difference at the .01 level. Similarly, the idea that a high count is a disadvantage to the dealer can be seen in the "Counter" player's data. As previously mentioned, this player plays with the dealer's strategy and records its winning rate when the count is both low and high. This player won 1.1% fewer games when the count was high than when it was low. Once again, this is statistically significant at the .01 level.

Upon examining the learning rate of the two agents, some other conclusions can be made. A graph of the winning percentage over time for both models:



First, both models' win percentages increase greatly and then asymptotically level off at some point. This to be expected because the models correct themselves upon losses. As they lose less, there will be fewer updates. Also, once the all of the stored probabilities reach the true probabilities of being able to hit without busting, the frequency of increasing and decreasing the

individual probabilities is reduced. When the stored probabilities stabilize, the win percentage of the model stabilizes.

Another interesting observation is that the CountingLearner agent learns at a slower rate than the Learner agent. This is expected as well. The CountingLearner agent has three times as many values to learn as the Learner agent. Since only one value is updated per hand, it should take considerably longer for the CountingLearner to reach a stable state.

6. Conclusion

Based on its results, this project turned out to be successful. The goal of creating an agent that can learn how to play blackjack was accomplished. Unfortunately, the learned strategy could not best the dealer's simple strategy, but it is a significant improvement on the basic strategy the learning algorithm begins with. An improvement going forward could possibly be to change how the algorithm updates an agent's probability values. An original attempt had these values being updated to the fraction of times a player didn't bust when hitting at that value. This idea was abandoned early on because if an agent with a relatively safe hand, say a hand summing to 12, busts on its first attempt to hit, the agent would never hit on 12 again. Also, this algorithm doesn't take into account the dealer's card. Perhaps more success could be found in the future by looking into ways to blend this abandoned idea with the current algorithm.

Another interesting possible attempt at blackjack AI going forward is to add functionality for betting. The agent can try to maximize its monetary earnings rather than its winning percentage. This could take a considerable amount of additional work, as other aspects of blackjack like splitting a hand and doubling down would need to be accommodated. Also, a

currency system would need to be created. Hopefully, combined with the current AI, the algorithm could learn when to bet more, and come close to breaking even financially.

Despite the possible improvements and expansions to this project, the current results are substantial. The algorithm starts with a rule that guesses about whether to hit or stand, and has a win percentage of less than 30%. Over thousands of hands of blackjack, the agent improves its strategy upon each loss, and eventually wins at a rate of over 38.5%. There are two agents: one that can count cards, and one that cannot. Results from this project allow for evaluation of the effectiveness of expanding the state of the game to include the deck – another of the project's goals. Overall, many ideas surrounding both Artificial Intelligence and blackjack have been explored, and the project has been a success.