A Q-Learning Approach to Successful Blackjack Play

Amy Kang, George Zhang
CS182, Harvard University, Fall 2016
https://github.com/gzhang01/cs182FinalProject

Our goal was to investigate how well a reinforcement learning agent, specifically a Q-learning agent, could do in learning to play blackjack. We started with a simple hit/stand simulation, before expanding to include doubling. We also looked at the effects strategic betting and card counting using both the Kelly criterion and the MIT betting strategy. In the first two instances, we used a random agent and an agent playing the basic strategy as benchmarks to compare against, while in the strategic betting trials, we compared agents with betting strategies to agents betting uniformly. In each instance, we monitored the win rate of the agent as well as the survival time, our proxy for monetary success. We found that our Q-learning agent consistently did better than the random agent but failed to match or beat the basic strategy agent. In addition, we found that strategic betting with the Kelly Criterion produced worse results than constant betting, but the MIT strategy produced comparable results.

1 Introduction

Blackjack is a popular casino game with a very simple premise: given two cards out of a standard card deck and knowledge about one of the dealer's cards, act optimally to beat the dealer without busting. Despite its apparent simplicity, the stochastic nature of the game makes predicting outcomes very difficult; from a given hand, choosing to hit can lead you to ten other hand values with varying probabilities based on the cards remaining in the deck. In addition, any action the player takes must be weighed against the dealer's card to determine the likelihood of winning.

While the underlying model of blackjack may be complicated, it is still one of the more beatable casino games, with a house edge of about 0.5%-1%, depending on the variation. Because of this, we would like to see exactly how close a player can come to beating the game. We believe that a reinforcement learning approach to learning is ideal, as we can learn the value of actions at a given state rather than solving for the best action, which avoids the model. Thus our RL agent must seek to both win often (have a high win rate) and bet intelligently (maximize payout on success and minimize loss on failure).

2 Background and Related Work

The best known blackjack strategy is the basic strategy. First introduced at IBM, the basic strategy is a policy over the player's and dealer's hand values [1]. In other words, given the player's hand value and the dealer's upcard, the basic strategy will output an action (out of either hit, stand, double, or split) that gives the player the highest likelihood of winning the round. The basic strategy was developed over millions of simulations, and thus represents optimal play [1]. In fact, following the basic strategy will reduce the house edge from about 5% against an unskilled player to around 0.5% [1].

Because the basic strategy chooses the optimal action based on likelihood of winning, we believe that improving the basic strategy would be difficult. Producing the basic strategy likely took more computing power than we have, and any solution based on simulations is likely to only approach the basic strategy and not improve it. Thus we decided to approach the problem from a different angle. The basic strategy seeks to maximize win percentage. We, instead, decided to try to maximize earnings. Note that if bets were constant, then these two problems are identical; if bets were constant, we could model our money as a random walk, which is maximized when the win percentage is maximized. However, we shall allow for nonconstant betting, which then creates two subproblems: maximizing win percentage and maximizing payout.

3 Problem Specification

The basic gameplay of blackjack is as follows:

1. The player and dealer are each dealt two cards. Both the player's cards are visible, whereas only one of the dealer's cards (the "upcard") is visible.
2. The player must decide whether to "hit" (take another card) or "stand" (to not take another card). The goal for the player is to beat the dealer's hand without going over 21 (known as a "bust"). Player bust leads to immediate loss of bet.
3. Once the player stands, the dealer then turns over her face-down card, and if the card is less than 17, must hit. Otherwise, she stands. If the dealer busts, she loses.
4. If the player's hand exceeds the dealer's, the dealer pays the player an amount equal to his bet. If equal, the player keeps his bet. If less, the player loses his bet.

More advanced gameplay involves the possibility to "double". Right after the cards are dealt, the player can decide to double his bet in exchange for one more card. This means that the player cannot decide to double after previously hitting and the player cannot receive any more cards after deciding to double.

More variations exist, with numerous other possible actions given certain conditions, but we shall limit our analysis to those described above. Regardless of the variation, blackjack can be summarized as a Markov decision process (MDP). As an MDP, the state would be the set of all cards seen thus far. In our case, however, we will model each state as a (player hand value, soft, dealer hand value) tuple, where soft describes whether or not the player's hand contains an ace. Note that an ace can be valued either as a 1 or 11, making busting with an ace in hand less likely. In this case, our model is not strictly an MDP[1], as it no longer truely maintains the Markov property, but since we are taking an RL approach and not solving it rigorously, and since the transition probabilities change very little when playing an 8-deck game, this distinction is not as important. Transitions between these states are stochastic; if the player or dealer decide to hit, the next state is determined by the value of the dealt card. A stand decision or a bust deterministically sends the game to a terminal state, where payoff can be calculated. Doubling can be represented the same way, as the bet amount can be external to the MDP itself, used only when calculating payoff.

4 Approach

Because we need a platform on which to train and test our agent, we started by building a blackjack simulator. The simulator itself exists in game.py. We then decided that a good benchmark for any learning agent we built would be a random agent, which simply places uniform bets of $10 and chooses actions randomly. We would expect any learning agent to be able to do better than random. We also built a basic strategy agent, which places uniform bets of $10 and decides on an action according to the basic strategy.

For our own learner, we used a modified Q-learning algorithm. The algorithm can be given as follows:

Algorithm 1 Q-Learning Agent

```
procedure UPDATE(state, action, reward, next)
    if reward = None then
        reward ← max_a Q(next, a)
    end if
    Q(state, action) ← Q(state, action) + α× (reward - Q(state, action))
end procedure
```

We decided to take a Q-learning approach to this problem for several reasons. First, Blackjack can be summarized as an MDP and offers an intuitive action-reward model where rewards are dependent on

[1]It is actually a POMDP, with the set of all cards seen as the hidden states and the hands produced as the observable emissions.

the values of the player's bets. Second, due to the stochastic and non-adversarial nature of the game, it seems ill-suited for approaches that search the state space. Finally, because it has a relatively small state space, successfully training a Q-learning agent remains feasible, while searching through the game tree is a difficult, exponential task.

We made one main modification to the basic Q-learning algorithm [4]. Because discounting future rewards does not really make any sense in blackjack (getting a 20 after 2 hits should not be worth less than getting a 20 after 1 hit), we wanted to take out the discount factor. We could have done this by simply setting γ equal to 1, but we decided a more intuitive way of thinking about this change is to simply let the reward be the value of the next state whenever the round has not yet ended. In the case where the round has ended, the reward is then just our payout. In this way, we can think of our reward as either the payout we got from the round or the "temporary payout" from going to a better state. Notice that our update rule is exactly the same as the basic Q-learning algorithm if we allow reward to be zero whenever the round has not yet ended and allow γ to be 1, but we decided that this structure made more sense for our use case.

We also utilized a ϵ-greedy strategy for obtaining our actions while training. Our implementation was very straightforward and we did not modify the algorithm at all. The pseudocode for the algorithm is shown below:

Algorithm 2 ϵ-greedy

```
procedure GETACTION(state, actions)
    with probability ε: return random action from actions
    return getActionFromPolicy(state, action)
end procedure
```

With the ϵ-greedy algorithm, we could prioritize exploration during the training trials, thus gathering as much data as we could. In addition, we could switch to a purely exploitative strategy when testing by setting $\epsilon = 0$, which was motivated by the fact that the basic strategy should be followed strictly for optimal gameplay [1].

We also experimented with a softmax action selection method in both the training and testing phases. Previously, when our Q-learning agent was choosing the action with the highest Q-value, we noticed that differences in policy between the policy learned by our agent and the basic strategy were often attributed to small differences in Q-value (less than 0.2), but our agent would deterministically pick the disadvantageous action (according to the basic strategy) with the greater Q-value. With a softmax approach, we could allow our agent to sample an action a from its available actions at state s with probability according to the function:

$$P_s(a) = \frac{e^{Q(s,a)}}{\sum_i e^{Q(s,i)}}$$

The pseudocode for the algorithm is shown below:

Algorithm 3 Softmax

```
procedure SOFTMAX(state, actions)
    prob ← []
    for i ∈ [0, len(actions)) do
        prob[actions[i]] ← exp{Q(state, actions[i])}
    end for
    denom ← sum(prob)
    for i ∈ [0, len(prob)) do
        prob[i] ← prob[i] / denom
    end for
    return actions[i] with probability prob[i]
end procedure
```

Finally, for the strategic betting portion, we decided to investigate two betting strategies: the Kelly Criterion and the MIT betting strategy. The Kelly Criterion calculates the optimal bet as a percentage of the total cash on hand [3]. It aims to maximize the utility of wealth rather than wealth itself, assuming a model of diminishing marginal returns [3]. For our simulation, the percentage bet is equal to 0.05 times the true count[2]. While the Kelly Criterion provides a general betting strategy (and is thus applicable not only to blackjack but also to various other gambling scenarios such as the stock market), the MIT betting strategy was formulated specifically for blackjack. In the MIT betting strategy, a basic unit is set, and each bet is calculated by multiplying the basic unit by one less than the true count [2]. If this bet is ever below the betting minimum, the minimum is bet [2]. Thus we could compare these two strategies with each other as well as gameplay with these strategies to gameplay without.

Of course, in order to strategically bet, we need to be able to count cards. We took a very simple card counting algorithm, where a card with value 2 through 6 is assigned a +1, a card with value 10 through A is assigned a -1, and a card with value 7 through 9 is assigned a 0. The counts are summed as we see the cards, producing a total count. Positive counts are advantageous to the player, since this means fewer low cards are remaining in the deck, thus increasing the likelihood of blackjack and dealer bust. To capitalize on card count, we tried expanding our state space of our Q-learner to include the count.

One caveat to our approach is that instead of looking directly at amount of money, to aggregate our data, we looked instead at number of rounds before bankrupcy (which we termed the survival time). We believed that because gambling is self-reinforcing (that is, winning makes people want to play more), a player is unlikely to magically stop at some optimum. In addition, because gambling games can easily be modeled as random walks, we know that survival time can serve as a proxy for how well a player is doing in terms of monetary gain. As a result, we decided to look at survival time rather than some metric of money in our analysis.

5 Experiments and Results

Because the target outcome of any given state is unknown (i.e. at any point in the game, there is no "correct" action), we cannot test our agent against a dataset of expected outcomes. We can, however, compare the performance and policy of our agent against that of other agents. In our analysis, we'll be comparing the performance of our Q-learning agent against the random agent and the basic strategy agent. Performance is measured by running our blackjack simulator and having our agents play rounds until they go bankrupt. We can then calculate their win percentage and the number of rounds before bankrupcy.

5.1 Hit and Stand

We started by only allowing the agents to hit and stand. These two actions represent the most basic gameplay for blackjack, and so this version provided a good starting point. In addition, with fewer actions, our state space was smaller, and so we were able to iterate on design while spending little time on training.

	Win Rate	Survival Time (rounds)
Random Agent	29.55%	290.32
Basic Strategy Agent	43.25%	4089.31
Q-Learning Agent (max) [3]	33.23%	2282.54
Q-Learning Agent (softmax)	33.10%	2189.65

Table 1: Win Rates and Survival Times for Three Blackjack Agents in Basic Hit/Stand Gameplay

[2]The true count is equal to the count divided by the number of decks left.

[3]Q-Learning agent was trained with $\epsilon = 0.8$ and $\alpha = 0.1$ over 1 million rounds. Results shown are for testing data. Note that during testing, ϵ was reset to zero to exploit learned policy. The Q-learning agent with softmax was trained with the same learning rate and number of rounds.

The results we obtained are in table 1. The win rate was found by calculating the total number of rounds won over 100 games[4]and dividing by the total number of rounds played. The survival time is the average survival time over the 100 games. Note that in this and every trial afterward, agents started with $1000 initially and made constant bets of $10, unless otherwise specified (esp. see Betting Strategies).

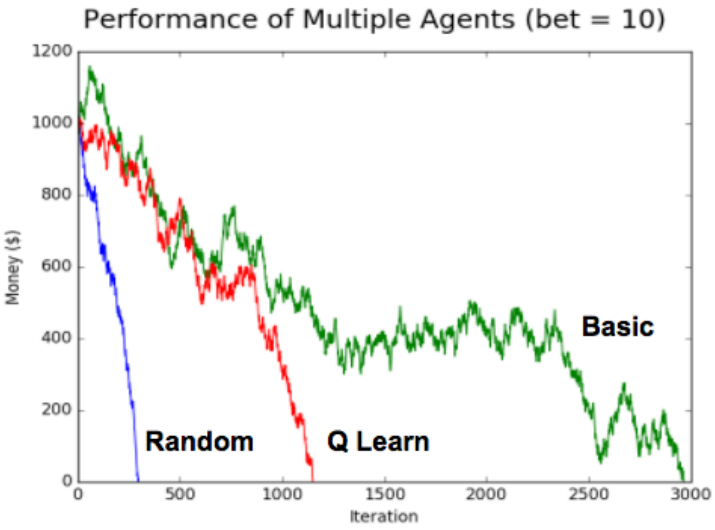


Figure 1: Money Over Iterations for Blackjack Agents

We can see that our Q-learning agent is better than the random agent in terms of both win rate and survival time, though it does not quite reach the performance of the basic strategy agent. We can also see that our softmax strategy did marginally worse than our max strategy. Because of this, and since the basic strategy should be followed rigorously, we decided to stick with the max decision-making paradigm. We continued by investigating the discrepancy in survival time, which we did by graphing the agent's money over training iterations. An example game is shown in figure 1. Note that for this and all future usages of our Q-learner, we maintained the max decision-making.

Player Hand	Dealer Hand									
Hard	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
12	H	H	S	S	S	H	H	H	H	H
11	H	H	H	H	H	H	H	H	H	H
10	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H
5	H	H	H	H	H	H	H	H	H	H
4	H	H	H	H	H	H	H	H	H	H
Soft	2	3	4	5	6	7	8	9	10	A
A, 9	S	S	S	S	S	S	S	S	S	S
A, 8	S	S	S	S	S	S	S	S	S	S
A, 7	S	S	S	S	S	S	S	H	H	H
A, 6	H	H	H	H	H	H	H	H	H	H
A, 5	H	H	H	H	H	H	H	H	H	H
A, 4	H	H	H	H	H	H	H	H	H	H
A, 3	H	H	H	H	H	H	H	H	H	H
A, 2	H	H	H	H	H	H	H	H	H	H
A, A	H	H	H	H	H	H	H	H	H	H

(a) Basic Strategy Actions

Player Hand	Dealer Hand									
Hard	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	H	S
17	S	S	S	S	S	S	H	S	S	S
16	S	S	N	S	H	H	H	H	S	H
15	S	N	S	S	S	S	S	H	H	H
14	S	S	S	S	S	S	H	H	H	H
13	S	S	H	S	S	H	H	H	H	H
12	S	H	S	S	H	H	H	H	H	H
11	H	H	H	H	H	H	H	H	H	H
10	H	H	H	H	H	H	H	H	H	H
9	H	H	H	H	H	H	H	H	H	H
8	H	H	H	N	N	H	H	H	H	H
7	H	H	S	H	H	H	H	H	H	H
6	H	S	N	H	H	H	H	H	H	H
5	S	H	H	H	N	H	H	H	H	H
4	H	H	S	H	H	N	H	H	H	H
Soft	2	3	4	5	6	7	8	9	10	A
A, 9	S	S	S	S	S	S	S	S	S	S
A, 8	S	N	S	S	S	S	S	S	S	S
A, 7	S	S	S	H	S	H	S	S	S	S
A, 6	H	H	N	S	H	H	H	H	H	H
A, 5	S	S	H	H	H	H	H	H	H	H
A, 4	H	H	H	H	S	H	H	H	H	H
A, 3	H	N	H	H	H	H	H	H	H	H
A, 2	H	H	H	H	H	H	H	H	H	H
A, A	S	H	H	H	H	H	H	H	H	H

(b) Q-Learning Actions

Figure 2: Comparison of Actions Between Basic Strategy and Q-Learner in Basic Gameplay[5]

[4]A game is defined to be the set of rounds that the agent plays ending at bankrupcy.

[5]An entry of "N" represents a state where the Q values for the different actions are too similar to be significant (absolute difference < 0.2)

From this, we noticed that our Q-learning agent was actually doing relatively well initially, but then ran into a series of poor decisions around iteration 1300 to 1600. In order to fully understand this decline, as well as to determine why our win rate was lower, we decided to look at the actions associated with the Q-learning policy, which is shown in figure 2b.

Here, we can see the discrepancy between our agent and the basic agent. While the policy learned looks similar to the basic strategy for the most part, there are definitely differences. Actions when the player hand is between a value of 12 and 16 appear to be very noisy. While unfortunate, this result is not unexpected, as anyone who has played blackjack knows that having a hand value of 12-16 is difficult. While we would expect our Q-learning agent to eventually approach the basic strategy, we accept that this range of values may be difficult for a reinforcement learning agent to learn.

The lower-left corner of the hard values necessitates a different analysis. Since getting a player hand value of 5 is relatively unlikely (we would need a 2 and a 3 in order to do this), combinations at this level likely do not occur that often. As a result, we expect this difference to be due more to lack of training data in this region. We attribute the variation in the soft values to this issue as well, since pairing an ace with these values occurs infrequently. Unfortunately, since the nature of the game is stochastic, we felt that hard-coding training iterations for these regions was a poor solution, and so we let the results stand.

5.2 Double

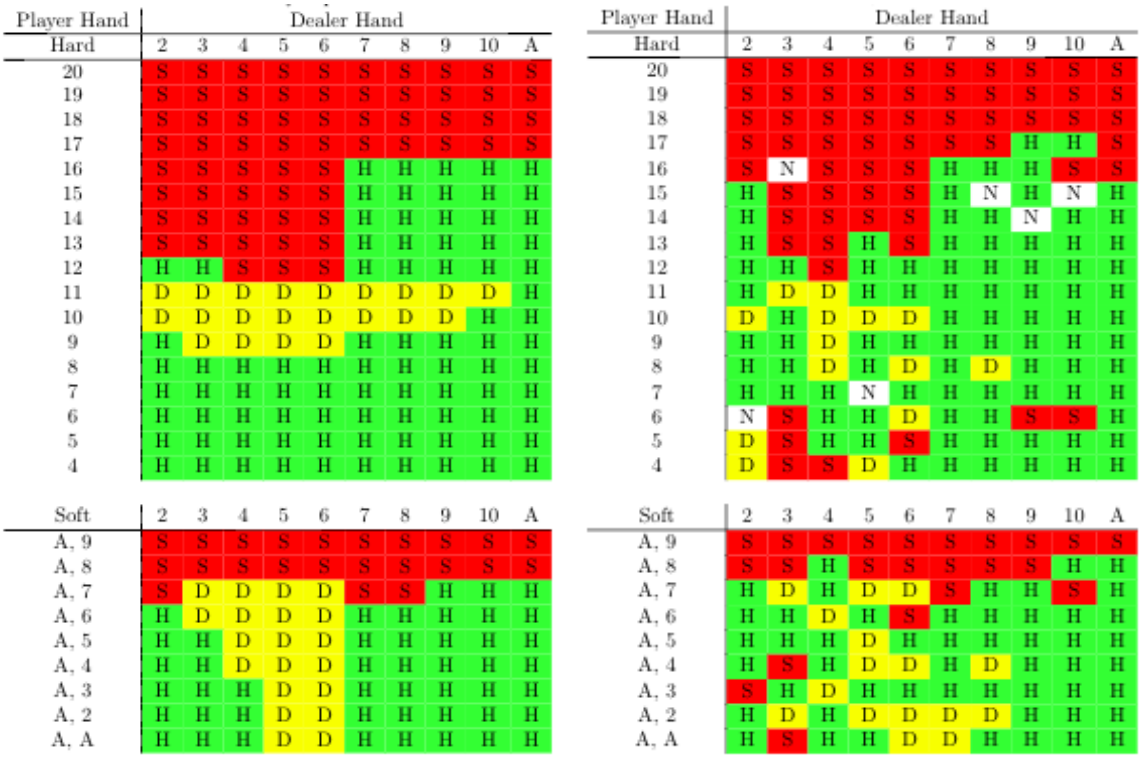
Our next step was to expand the game to include doubling. We expect that strategic doubling will allow us to take advantage of good hand values, and thus bet and potentially win more. As a result, this should allow us to remain in the game longer before we go bankrupt. Indeed, the basic strategy expanded to including doubling shows this exact behavior, as seen in table 2. The win rate is exactly the same, yet the agent survived for more than twice as long. Our Q-learning agent, however, was not as successful. While still better than the random agent, the Q-learner survived on average 300 rounds fewer. To investigate why, we again constructed their action tables (Fig. 3).

	Win Rate	Survival Time (rounds)
Random Agent	30.27%	222.06
Basic Strategy Agent	43.25%	8975.56
Q-Learning Agent	36.62%	1961.18

Table 2: Win Rates and Survival Times for Three Blackjack Agents in Expanded Doubling Gameplay

We can see that Q-learner action table is much noisier than we might like. Even though we ran the learner for one million iterations, it appears that with the expansion of the state space that comes with including doubling, we do not get enough data to adequately produce a reliable policy. This issue is likely accentuated by the fact that we can only double on the first action, and so we are able to learn much fewer iterations on doubling than either hitting or standing. When we add in the fact that certain combinations are very unlikely to been seen anyway (lower-left corner of hard values), we see that the learner ultimately does a poor job.

While we attribute this noisiness to a lack of training data, it is still possible that there is another underlying cause. We have tried training for more iterations, but our preliminary tests did not prove fruitful. If it is solely a training data issue, we believe that the number of times we need to experience these states to get accurate information is too high considering the likelihood of landing in them, and so the training time would be much too high. Thus we turned our focus to strategic betting instead of trying to approach the basic strategy with doubling.



Player Hand	Dealer Hand									
Hard	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S
16	S	S	S	S	S	H	H	H	H	H
15	S	S	S	S	S	H	H	H	H	H
14	S	S	S	S	S	H	H	H	H	H
13	S	S	S	S	S	H	H	H	H	H
12	H	H	S	S	S	H	H	H	H	H
11	D	D	D	D	D	D	D	D	D	H
10	D	D	D	D	D	D	D	D	H	H
9	H	D	D	D	D	H	H	H	H	H
8	H	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H
5	H	H	H	H	H	H	H	H	H	H
4	H	H	H	H	H	H	H	H	H	H
Soft	2	3	4	5	6	7	8	9	10	A
A, 9	S	S	S	S	S	S	S	S	S	S
A, 8	S	S	S	S	S	S	S	S	S	S
A, 7	S	D	D	D	D	S	S	H	H	H
A, 6	H	D	D	D	D	H	H	H	H	H
A, 5	H	H	D	D	D	H	H	H	H	H
A, 4	H	H	D	D	D	H	H	H	H	H
A, 3	H	H	H	D	D	H	H	H	H	H
A, 2	H	H	H	D	D	H	H	H	H	H
A, A	H	H	H	D	D	H	H	H	H	H

(a) Basic Strategy Actions

Player Hand	Dealer Hand									
Hard	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
17	S	S	S	S	S	S	S	H	H	S
16	S	N	S	S	S	H	H	H	S	S
15	H	S	S	S	S	H	N	H	N	H
14	H	S	S	S	S	H	H	N	H	H
13	H	S	S	H	S	H	H	H	H	H
12	H	H	S	H	H	H	H	H	H	H
11	H	D	D	H	H	H	H	H	H	H
10	D	H	D	D	D	H	H	H	H	H
9	H	H	D	H	H	H	H	H	H	H
8	H	H	D	H	D	H	D	H	H	H
7	H	H	H	N	H	H	H	H	H	H
6	N	S	H	H	D	H	H	S	S	H
5	D	S	H	H	S	H	H	H	H	H
4	D	S	S	D	H	H	H	H	H	H
Soft	2	3	4	5	6	7	8	9	10	A
A, 9	S	S	S	S	S	S	S	S	S	S
A, 8	S	S	H	S	S	S	S	S	H	H
A, 7	H	D	H	D	D	S	H	H	S	H
A, 6	H	H	D	H	S	H	H	H	H	H
A, 5	H	H	H	D	H	H	H	H	H	H
A, 4	H	S	H	D	D	H	D	H	H	H
A, 3	S	H	D	H	H	H	H	H	H	H
A, 2	H	D	H	D	D	D	D	H	H	H
A, A	H	S	H	H	D	D	H	H	H	H

(b) Q-Learning Actions

Figure 3: Comparison of Actions Between Basic Strategy and Q-Learner in Expanded Gameplay

5.3 Strategic Betting and Counting

We initially wanted to include the count in two aspects of our Q-learner: in placing bets and in taking actions. However, we found that including the count in the state space (i.e. incorporating count into actions) failed to increase survival time. Using the same learning rate, epsilon, and number of training rounds as the Q-learner sans count, we saw a small increase in win rate (34.53%) but a fairly large decrease in survival time (1142.02 rounds) over 100 trials. Because this failed to increase survival time, we decided to move forward with our original Q-learner, which did not include the card count, and simply explore the effect of incorporating count with betting strategy.

We decided to look at two strategies: the Kelly Criterion and the MIT betting strategy. Previous papers have suggested that the Kelly Criterion provides benefits in the long run, but do not show much improvement over random betting in the short run [3]. However, this is under the assumption that the probability of winning is greater than the probability of losing. Because blackjack has a positive house edge, this assumption is not true, and so the Kelly Criteron may actually cause our agents to lose faster than they would normally. As a result, we also wanted to test the MIT betting strategy, which was specifically designed for blackjack. The results of our simulations are in table 3. Note that because

	Betting Strategy	Win Rate	Survival Time (rounds)
Basic Strategy	Constant Betting	43.25%	4089.31
	Kelly Criterion	43.39%	1190.71
	MIT	43.39%	4136.66
Q-Learning	Constant Betting	33.23%	2282.54
	Kelly Criterion	33.04%	358.06
	MIT	34.72%	922.37

Table 3: Effect of Different Betting Strategies on Win Rate and Survival Time (min bet: $10)

doubling did not work as well as desired, both our benchmarks and our betting agents played on the simple hit/stand version.

We notice that the Kelly Criterion does indeed negatively impact our survival time for both the basic strategy and the Q-learner. The MIT strategy appears to have little effect on the basic strategy, but also decreases survival time for the Q-learner. We suspect this discrepancy may be due to the lower win rate for our Q-learner in relation to the basic strategy. Since our actions are not optimal, increasing our bet is likely hurting us more than it is helping, even when the true count is high, thus causing us to go bankrupt more quickly. Our guess is that if we were to allow our Q-learning agent to decrease its bet below the uniform bet ($10) in order to play more conservatively when the count is disadvantageous, we would be able to increase our survival time.

	Starting Amount	Kelly Criterion		MIT	
		WR	ST	WR	ST
Basic Strategy	$1,000	43.39%	1190.71	43.39%	4136.66
	$10,000	43.31%	2156.85	43.31%	40146.07
	$100,000	43.42%	3145.09	—	—
	$1,000,000	43.35%	3869.50	—	—
Q-Learning[7]	$1,000	33.04%	358.06	34.72%	922.37
	$10,000	32.73%	499.92	36.95%	6267.66
	$100,000	32.12%	612.52	—	—
	$1,000,000	31.81%	748.18	—	—

Table 4: Effect of Starting Amount of Money on Win Rate (WR) and Survival Time (ST, in rounds)

Due to the difference between short term and long term behavior in the Kelly Criterion, we also investigated what happens when we drastically increase our initial bet (Table 4). By doing this, we expect our simulation to run for longer, and so we can see some long term trends. We notice that starting amount is more or less logarithmically proportional to survival time using the Kelly Criterion, whereas it's roughly linearly proportional using the MIT strategy for the basic strategy agent. We suspect that this is because the Kelly Criterion bets a percentage of money, so an increased starting amount means a similarly increased bet size, which leads to quicker bankrupcy. The MIT strategy, however, maintains the same basic unit, and so we see a proportional increase. The Kelly Criterion behavior is roughly replicated in our results, though the linear behavior for the MIT strategy is not. We suspect this is because our low win rate weights us more toward losing, and thus leads to bankrupcy faster than expected.

6 Discussion

Our original goal and conjecture was that the policy learned by our Q-learning agent would converge to the policy described by the basic strategy. As our results show, this did not happen as well as we expected. While many of the learned actions matched the basic policy, there existed certain regions of the state space where learning was apparently difficult (Fig. 2). This noise was made worse when we added an additional action (Fig. 3). Because adding another action expanded our space of Q-values by 50%, we concluded that many of these differences could be attributed to a lack of training information, especially in the less probable states. Given more time and computing power, we would have liked to train for many more iterations and get more trials in these less probable states to see if our results will indeed converge to the basic strategy. It would be interesting if it does not, as this would suggest the Q-learning approach does not produce the mathematically optimal action in this game. One possibility to address this issue that avoids too much additional training is by using a Monte Carlo simulation, whereby the rewards are propagated back to all actions on every iteration. In this way, it will take fewer iterations to properly learn the states with lower values, which make up most of our ambiguous states.

We also found that, due to its stochastic nature, blackjack is a difficult game to simulate and produce replicable results. While our win rate and average survival times do indeed suggest that our Q-learner fell short of the basic strategy (Table 1), Figure 1 shows that our Q-learner could play on par with the

basic strategy agent over certain periods of time. It is possible that over this period, we somehow got card values that fell into the less noisy regions of the state space, but it is more likely that the stochasticity of blackjack produced the comparable results. The question then is whether the dramatic fall in cash over iteration 900-1100 might also be due to stochasticity or our learned actions. While we do not debate that our Q-learner ultimately performed worse on average, it appears that data from one single simulation is nowhere near reliable as a performance metric.

Our doubling results did confirm our belief that strategic use of doubling may result in greater monetary gain as measured by survival time (see Table: 2). The basic strategy with doubling survived more than twice as long on average than the basic strategy without doubling, despite having a nearly identical win rate. This suggests that the longer survival time was indeed the result of winning more money, and not related to simply winning more often. Unfortunately, we could not reproduce this behavior with our Q-learner, though we think that more training could resolve this issue.

While we believed that strategic betting could increase earnings over the long run, our results do not support this hypothesis. Betting using the Kelly Criterion actually reduced our survival time significantly, while the MIT betting strategy had very little effect for the basic strategy agent (see Table 3). This is a curious result, as conventional wisdom suggests that when the deck is more favorable to the player, betting high should result in greater gain. Further investigation into this area could proceed in two directions: investigation of alternative strategies or alteration of the minimum bet. In regards to the former, we believe that variations to the MIT strategy that drastically increase the bet when the count is very high may produce more desirable results, as a linear scaling based on count may not be able to take advantage of hot decks. Another variation may be to use a MIT strategy below some true count and transition to the Kelly Criterion above it, which would also take advantage of very high counts. Regarding the latter, our minimum bet was set to \$10, which was equal to our bet in the constant betting strategy. Thus our strategic betters were able to take advantage of higher counts, but could not insulate themselves against losses when counts were low or negative. Decreasing the minimum bet may allow the agent to play more conservatively when the count is low, and thus allow the agent to exploit both good and poor counts.

One final area for future investigation pertains to the data itself. We used survival time as a proxy for monetary gain, as it was an easy statistic to aggregate and collect. While less realistic[6], peak cash amount may also be an interesting metric to investigate. We think strategic betting may produce a higher average global optimum, but further investigation is needed in this area as well.

Variations on our original Q-learner seemed limited in improving our performance in part due to the low win rate. It would be interesting to see whether these same strategies to reduce the house edge, particularly expanding the state space to include card count, would prove more effective on a learning agent with higher base win rate. Indeed, the only time a player should deviate from the basic strategy is when advantageous gameplay (such as card counting) suggest it may be a good idea [1]. As a result, we believe that if our Q-learner could be trained to converge to the basic strategy at zero count, then any deviations from the basic strategy at other counts will constitute an advantageous deviation from the basic strategy, and thus a combination of counting (as well as betting strategies that capitalize on the count advantage) and higher win rate could produce a longer survival time. Although our Q-learning agent could not beat the basic strategy agent, we are optimistic that a reinforcement learning agent has the potential to do so.

[6]Less realistic only in the sense that a human player is unlikely to stop at exactly the global optimum.

A System Description

Our system can be divided into three parts: simulation, automated agents, and data collection. All simulation and agent components lie in the **blackjack** folder while data collection components (including any .csv files and graphs, but excluding gatherData.py, which requires interfacing with the simulation and agents) lie in the **data** folder.

A.1 Simulation

The simulation itself is run on **game.py**, which houses the Blackjack class and main, which parses any given flags, creates a player, and launches the game (if the Q-learning flag is passed in, main also handles the training and testing cycles for our agent). Note that the version of the codebase on master includes doubling as an available action. Available flags, as listed in our README.md, are:

- -a [agent], -agent [agent]
  This runs the selected agent, where available agents include 'random', 'basic', and 'qlearning'. The default agent is 'player', which is a user-controlled agent.
- -cd, -collectData
  Collects data on money per round.
- -f [file], -file [file]
  Saves collected data to file where the default file is "data/default.csv"
- -m [amount], -money [amount]
  Sets the starting money for agent to 'amount', where the default is $1,000.
- -np, -noPrint
  Doesn't print anything during program execution (excepting number of training rounds completed in Q-learning).
- -qiter [iterations]
  Trains the Q-learning agent on 'iterations' rounds, where the default value is $10,000$.

The Blackjack class depends on the Card class (in **card.py**), Deck class (in **deck.py**), Constants class (in **constants.py**), and the Player class (in **player.py**).

A.2 Agents

Fields and methods necessary to the player are located in the Player class in the **player.py** file. This includes the player's hand and bankroll, as well as methods to allow the player (user-controlled) to choose their bet and action. Helper functions to allow the game to add cards to the player's hand, retrieve the player's hand value, and discard the player's hand are also housed here. The Player class is the root class for all of our agents.

The Automated Player class, located in the **automatedPlayer.py** file, inherits from the Player class and overwrites the method to choose the bet to return a uniform bet (defined in **constants.py**) of $10. Our random (in **randomAgent.py**, basic strategy (in **basicStrategyAgent.py**), and Q-learning (in **qLearningAgent.py**) agents inherit from the Automated Player class, overwriting the methods choose bet and choose action according the descriptions provided in our approach. In order to change the betting strategy of either the Q-learning agent or the basic strategy agent, we can change the function called by chooseBet to either the uniform betting strategy, the Kelly Criterion, or the MIT strategy.

A.3 Data Collection

Using the -cd flag when running game saves the relevant .csv file (either named with the -f flag or default.csv) in the data folder. Note that the .csv suffix is unecessary when specifying file name here. For example:

```
python game.py -a qlearning -np -cd -f qlearningData
```

In order the generate line graphs representing the player's bankroll over rounds of game play, we can either run:

python plotData.py [file]

for a single file, or, run:

python plotMultiple.py [file 1] [file 2] [file 3] [destination]

to plot the performance of multiple agents.

In order to attain aggregate data (namely average win rate, average survival time, average optimum bankroll, and standard deviation of both survival time and optimum bankroll) over 100 trials, we ran **gatherData.py**, commenting out the irrelevant sections. Functionality to gather data on both the random and basic strategy agents, train and test the Q-learning action, and output both the Q-learner's policy and Q-values to a .csv file are included in **gatherData.py**.

B Group Makeup

In the initial phase of the project, George built the basic simulator with options to hit and stand. Amy built the automated agents on top of this simulator, including the random, basic strategy, and initial Q-learning agent. Both members contributed to debugging and refining the Q-learning agent to reach the performance discussed in this report.

In the later phase of the project, George built the infrastructure necessary to gather data and plot data (namely **gatherData.py** and **plotData.py**) while Amy expanded the simulator to include doubling and the agents to include other betting strategies. George also worked on expanding the state space to include counting (functionality which was used to expand to other betting strategies) and experimenting with the softmax action selection rule. Both members contributed to the collection of data, discussion, and analysis of the results.

For a more detailed breakdown of how work was split up, please feel free to look on Github[7]:
https://github.com/gzhang01/cs182FinalProject

References

[1] https://www.cs.bu.edu/~hwxi/academic/courses/CS320/Spring02/assignments/06/basic-strategy.html.

[2] http://www.blackjack-trainer.net/blackjack-betting-strategy/.

[3] Jane Hung. Betting with the kelly criterion. 2010. Available at: https://www.math.washington.edu/~morrow/336_10/papers/jane.pdf.

[4] Andrew G Barto Sutton, Richard S. Reinforcement learning: An introduction. 1998.

[7] We know you want this for grading purposes, but we feel like we collaborated for much of the project and it is difficult to separate our work into disjoint sets. :)