Playing Blackjack using Q-Learning and Neural Networks

Christen Ford

March 2, 2021

1 Introduction

This report details the design and implementation of a stochastic blackjack agent built around Q-Learning and Neural Network principles. Blackjack is a card game consisting of a dealer and a number of players where the goal is to have the highest hand total among all players (including the dealer) closest to twenty-one. A hand total exceeding twenty-one is a bust and a player with such a hand loses the round. Ties between players see the pot evenly split amongst all the players involved in the tie.

Blackjack is considered a stochastic card game because each player only knows the contents of their own hand plus one of the dealer's two cards (called the show card). Unlike Poker (a related card game) where players can bluff each other or otherwise sluice out what a player has in their hand, each player is engaged in a competition of pure luck. A player's initial hand sets the tone for the remainder of the round. Initial hand composition has a large psychological impact on whether a player chooses to hit or stand. The card a player receives on a hit can either make or break their hand. As such determining when to hit or stand can be modelled as a stochastic process. This paper will cover the background necessary to understand the model, the model itself, the implementation of the model and lastly, the results and conclusion.

This paper presents a combination of Q-Learning and Neural Network techniques rooted in probability theory implemented both as a mathematical model and Python implementation. The model itself utilizes expected payout and risk functions to determine actions as showcased in [Sil92]. initial results detail on average an 85% to 95% win rate out of 200 rounds per game.

2 Background

Blackjack is a card game with a dealer and a number of players. Each player tries to maximize the value of their hand such that it is as close to 21 as possible without going over. After all players have ended their turns, the dealer than takes their turn under the same rules. However, the dealer usually operates

under an additional rule wherein they stand on a certain hand value (usually 16 or 17). A trivial model for playing blackjack would simply leverage this rule. The rules of blackjack are simple.

Blackjack is usually played with a single deck in an amateur setting however, in a professional setting (i.e., a casino), multiple decks are used for a single game. A standard deck of cards consists of 52 cards of the suits following suits S:

$$S = \{Hearts, Clubs, Diamonds, Spades\} \tag{1}$$

and face values F:

$$F = \{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\} \tag{2}$$

Each suit consists of twelve cards of corresponding face-value. Under standard house rules, the J, Q and K count as a value of 10 while the A counts as a value of 11 unless it results in a bust. In this case, the A counts as a 1. This leads to an overall ruleset that is simple to understand and implement as well as develop into a model.

Q-Learning is a reinforcement learning technique based on Markov Decision Processes. Each state in a Q-Learning environment only depends on the prior state observed by the agent (the same as an MDP). What sets Q-Learning apart is that it extends the MDPs by implementing a state-action-reward paradigm based around Q-values. MDP's are based purely around developing a state transition table that states the probability of moving from state s to state s' under some assumption or set of assumptions. Q-Learning guides this process through the use of rewards as determined by the Bellman-Ford equation:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha * (r_t + \gamma * \arg\max_a \{Q(s_{t+1}, a)\} - Q(s_t, a_t)) \tag{3}$$

The above equation updates the Q-Value for a specified state and action at time t by weighting the expected reward against the estimated best future value. The above formula will eventually always converge to an optimal value as long as the state and action spaces are finite. α represents the learning rate and determine how quickly the agent responds to changes in the Q-Value. A value closer to 0 makes the agent disregard new information while a value closer to 1 makes the agent only consider new information and ignore past history. While γ is used to determine the importance of future rewards. A γ closer to 0 makes the agent shortsighted while a value closer to 1 makes the agent consider long-term goals.

Furthermore, Q-Learning is model free. MDP's require a known model to operate. Q-Learning does not. Q-Learning achieves it's goal by maximizing it's expected reward over all possible states generated from all possible actions from a current state. This allows Q-Learning to develop an optimal policy for any given MDP in a more efficient manner than a vanilla MDP would be able to.

A Neural Network is a family of semi-supervised/unsupervised machine learning algorithms focused on training a set of weights to produce a set of outcomes. Specifically, a neural network is a directed graph organized into an input layer, one or more hidden layers and an output layer. In modern neural

networks, input and output is always in the form of a vector. Vector operations are easy to implement in computer systems and they are efficient to work with from a mathematical perspective as the basics of Neural Networks are nothing more than Linear Algebra and Calculus.

Classical techniques to playing Blackjack such as those showcased in [Bal+56] and [TW73] have shown that the core fundamentals of Blackjack are easy to incorporate into a mathematical model. Furthermore, as shown in [Wu18], [Grawn], and [PS98] both Q-Learning and Neural Networks have already been applied to create agents capable of playing Blackjack. A combined approach to playing Blackjack that leverages both Q-Learning and Neural Networks should be feasible given a correct model design.

3 Model

3.1 The Probabilities of Blackjack

This section details my Blackjack model. This model takes into account all three stages of play in a standard Blackjack round {Betting, Playing, Evaluation}. During the Betting stage, all players decide whether to bet or leave. When I initially designed this model, I made it so that the agent would decide whether to continue playing or leave. After testing the model however, I concluded that this was not optimal for model training purposes as it made training sessions excessively short. As a result this feature did not make it into the final model.

During the Playing stage, the model generates an action distribution for hitting or standing that states the probabilities of each action. This model does not take into account splitting. For reference, a split in Blackjack occurs when during the initial draw of cards the player decides to split their hand into two separate hands whereby the extra hand places the same bet as the original hand. In Blackjack nomenclature, this is also referred to as doubling down. This action is not included in this model because the underlying implementation of the model would require non-trivial modification that I did not have time to account for.

Lastly, the Evaluation stage occurs when each player has concluded their turns and the dealer has drawn their cards. In this stage, hands are totaled and the winner(s) are chosen from amongst all players in the player pool. The model implements this stage as a mechanism for updating it's number of wins, losses and draws and overall chip delta. This information is utilized by the model as a part of wager, risk and expected payout calculations.

Note that research from [Bal+56] details the optimum strategy for playing Blackjack as well as mathematical proofs that verify their findings. However, I wanted to see if I could design something comparable. My model utilizes five related probabilities (some of which are detailed in [Dud15]) in it's calculations.

For a description of the symbols used in this section, please see Table 1.

1. $P(AB)$ - The probability that the agent will bust on next hit.

2. $P(A21)$ - The probability that the agent will have a blackjack on next hit.
3. $P(DB)$ - The probability that the dealer show and hidden card are a busting hand.
4. $P(D21)$ - The probability that the dealer show and hidden card are a blackjack.
5. $P(AT - DT > 0)$ - The probability that the agent's hand total is greater than the dealer's hand total.

Based on my analysis the prior probabilities are key to determining whether an agent wins or loses. I will now discuss each of these probabilities and their derivation in turn.

$$P(AB) = \frac{|ABC|}{|UC|} \tag{4}$$

$$P(A21) = \frac{|ABJC|}{|UC|} \tag{5}$$

$$P(D21) = \frac{|DBJC|}{|UC|} \tag{6}$$

$$P(AT - DT > 0) = \frac{|DLTC|}{|UC|} \tag{7}$$

This allows us to determine the probability of an agent winning a round:

$$\begin{aligned} P(AgentWinning) &= P(A21 \cup AT - DT > 0 \mid \neg AB \cap \neg D21) \\ &= \frac{P(A21 \cup AT - DT > 0,\ \neg AB \cap \neg D21)}{P(\neg AB,\ \neg D21)} \\ &= \frac{P(A21 \cup AT - DT > 0) * P(\neg AB) * P(\neg D21)}{P(\neg AB) * P(\neg D21)} \\ &= \frac{(P(A21) + P(AT - DT > 0)) * P(\neg AB) * P(\neg D21)}{P(\neg AB) * P(\neg D21)} \end{aligned} \tag{8}$$

3.2 Risk, Reward and Expected Payout

Prior to discussing risk, we need to talk about the softmax function that this model uses. This model utilizes a version of the softmax function for reinforcement learning:

$$\sigma(z)_i = \frac{\frac{e^{z_i}}{\tau}}{\sum_{j=1}^{K}(\frac{e^{z_j}}{\tau})} \tag{9}$$

Where τ is a temperature parameter that weights the contribution of each element of z. The larger the value for τ, the closer the resulting values in the distribution produced by softmax will be.

So, how does all of this help the agent determine actions? Let us look at how the model estimates risk.

$$Risk(s,a) = \begin{cases} \begin{cases} 1 - \frac{\Delta_{chips} - min(\Delta_{chips})}{max(\Delta_{chips}) - min(\Delta_{chips})}, \\ \quad \text{If } max(\Delta_{chips}) - min(\Delta_{chips}) > 0 \\ 0, \\ \quad otherwise \end{cases} & , s_r = 0 \\ \begin{cases} 1 - P(Agent\,Winning), \\ \quad \text{If } a = HIT \\ P(Agent\,Winning), \\ \quad \text{If } a = STAND \end{cases} & , s_r = 1 \\ \begin{cases} \frac{wins + draws + losses}{games\,played}, \\ \quad \text{If } games\,played > 0 \\ 0, \\ \quad otherwise \end{cases} & , s_r = 2 \end{cases} \tag{10}$$

In the above equation and those that follow s is a state, s' is a successor state, a is an action and s_r is the current game stage for a specified state.

Additionally, we need to look at how the agent estimates payoffs.

$$E|Payout(s,a)| = \begin{cases} risk(s,a) * s_{wager} & s_r = 0 \\ 200 * \frac{|A21C|}{|UC|} + \sum_{i=1}^{n}(200 * \frac{count(ABC_i^c)}{|U|}) & s_r = 1 \\ s_{\Delta_{chips}} & s_r = 2 \end{cases} \tag{11}$$

Of course Q-Learning also requires a reward function.

$$reward(s,a,s') = \begin{cases} reward1(s,a,s') & , s'_r = 2 \\ reward2(s,a,s') & , s'_r = 1 \\ reward3(s,a,s') & , s'_r = 0 \end{cases} \tag{12}$$

$$reward1(s,a,s') = \begin{cases} reward1\_1(s,a,s') & , s'_{outcome} = WIN \\ reward1\_2(s,a,s') & , s'_{outcome} = DRAW \\ 0 & , s'_{outcome} = LOSS \end{cases} \tag{13}$$

$$reward2(s,a,s') = \begin{cases} -250 & , a = HIT \\ 0 & , a = STAND \\ 0 & , otherwise \end{cases} \tag{14}$$

$$reward3(s, action, s') = \begin{cases} 100 & , wr = 0 \vee lr = 0 \\ base * wr + base * dr - base * lr & , otherwise \end{cases} \tag{15}$$

$$reward1\_1(s, a, s') = \begin{cases} 250 * risk(s, a) & , a = HIT \\ 250 & , a = STAND \\ 250 & , otherwise \end{cases} \tag{16}$$

$$reward1\_2(s, a, s') = \begin{cases} 500 & , s.r = 0 \\ (\frac{1 - softmax(s_r, s'_r)}{\sum softmax(s_r, s'_r))}) * 500 & , otherwise \end{cases} \tag{17}$$

3.3 Determining Wagers, Actions and Adjusting the Model

The model utilizes the following algorithm to determine wagers at the beginning of each round:

```
Algorithm determine_wager(state):
    num = current_chip_delta - min_chip_delta
    den = max_chip_delta - min_chip_delta
    if den > 0:
        risk = num / den
    else:
        risk = 0.5
    if risk <= 0.25
        wager = 10
    else if:
        wager = 20
    else if:
        wager = 50
    else:
        wager = 100
    while wager >= current_balance:
        decrease wager to next lowest wager
    return wager
```

With all of this information, we can finally discuss how the model determines actions. The model utilizes the following algorithm to generate action distributions:

```
Algorithm action_distribution(state):
    // PRE_ROUND_ACTIONS = {BET}
    // IN_ROUND_ACTIONS = {HIT, STAND}
    p = softmax(qtable[state].qvalues)
```

```
    if current game stage is PRE_ROUND then:
        zip PRE_ROUND_ACTIONS and p into a list of tuples
    else if current game stage is IN_ROUND then:
        zip IN_ROUND_ACTIONS and p into a list of tuples
    sort the list of tuples in descending order by probability
    return the sorted list
```

Furthermore, the model utilizes the following algorithm to determine actions:

```
Algorithm determine_action(state):
    action = NULL
    dist = action_distribution(state)
    p = random decimal in range [0, 1]
    for tuple in dist:
        a = tuple.action
        P = tuple.probability
        if p < P:
            action = a
            break
    if action is NULL:
        return dist[0].action
    return action
```

Note that Q-Values are determined using a modified version of the Bellman-Ford equation defined in the introduction:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha * (r_t + \gamma * Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{18}$$

In the modified Bellman-Ford equation, the model does not iterate every action. It simply performs a lookup operation on the Q-Value for the current action for the next state (the state at s_{t+1}).

The model maintains a weight and bias for every action at every stage of the round. These weights and biases correspond to a specific hash function calculated as: $state_hash = hash(current_hand, dealer_show)$. The state_hash acts as an index for the Q-Value table. At the end of each stage in a round the model will update it's weights and biases based on what happened during the round. The model uses the following equation to update weights.

$$weight_{s_t,a_t} = (1 - \beta) * weight_{s_t,a_t} + \beta * weight_{s_t,a_t} \tag{19}$$

And the model uses a similar formula to update biases.

$$bias_{s_t,a_t} = (1 - \delta) * bias_{s_t,a_t} + \delta * bias_{s_t+1,a_t} \tag{20}$$

The Q-Value for an individual state is thus determined by a linear function:

$$QUpdate(s_t, a_t) = weight_{s_t,a_t} * Q_{new}(s_t, a_t) + bias_{s_t,a_t} \tag{21}$$

When the model determines the action distribution, the Q-value for each action in the current game stage is looked up in the table and appended to a vector. This vector is then softmaxed and it's resulting values are paired with corresponding actions. This generates the action distribution. Note that the parameters utilized by the above equations can be found in Table 3.

This concludes the discussion on all of the components of the model.

4 Implementation

The actual Python implementation of this model is too large to list in this paper. Readers are directed to it's GitHub repository at: `https://github.com/gollum18/blackjack-agent`. Note that the implementation is split into four (5) primary files:

- /bqa/bqa/cards.py - Defines the Card and Deck classes. Implements playing card logic.
- /bqa/bqa/game.py - Defines Blackjack game logic and needed helper functions.
- /bqa/bqa/player.py - Defines Player, Agent and Dealer classes. The majority of this class is the implementation of this model.
- /bqa/bqa/qtable.py - Defines a dynamic Q-Table cache.
- /bqa/bin/game-runner - Defines a Python script for launching the test runner from the command line. Executables are built against this script to generate runnable versions of the Script that don't require calling the Python executable.

The implementation is written in Pure Python including the playing card logic. While there were certainly third-party libraries I could have used to bootstrap this project, I instead built every thing from scratch. The model implementation is entirely functional with no requirement for third-party libraries to be installed from PyPi.

Note that there are some quirks with this implementation:

- The current model and implementation does not account for splitting. This will be addressed in a future iteration of the model.
- There is *some* tight-coupling between classes however, the majority of the implementation is loosely coupled. I have not performed a thorough software analysis on the implementation but this will be addressed in future iterations.
- There are some code smells as well. Primarily unused code artifacts.
- There are no proofs for any of the mathematics defined by the model. I hope to address this in a future iteration though it may take some time as I am not an expert in proof writing.

- The determine_wager function is implemented incorrectly and I only caught it when I was writing this report. As it currently stands, the determine_wager function will determine a higher wager the higher the risk which is not something I intended. It should be the opposite. There was also a bug that I didn't catch when I looked over my code prior to submission in that the denominator for the determine_wager function was $\Delta_{max\_chips} - \Delta_{max\_chips}$ when it should be $\Delta_{max\_chips} - \Delta_{min\_chips}$ (per the model). This does not have any significant effect on the core performance of the model other than set the risk for the wager to 0.5 which in turn causes the agent to always wager 50 or less chips. This **will be** addressed in a future iteration of the implementation of the model.

The implementation is mostly documented. There are some additions that need put it, but it should act well as a guide to anyone wanting to extend this model or the implementation itself.

5 Results

Results I gathered from running the model with default parameters can be found in Table 2. Changing the parameters on the model will in fact impact the performance of the model. The implementation linked in the preceding section assumes sane defaults for the parameter models, however, I have not experimented with them enough to find optimal values for each. As this is a large model, it may be worth using a ML algorithm specifically designed to fine tune model parameters.

It is important to note that the accuracy of the model in all five tests only drops to 85%. Note that Blackjack has a large state space and one of the drawbacks of Q-Learning is that the larger the state space, the sparser the resulting Q-table. This has a negative side effect on model performance wherein the larger the Q-table, the less each Q-table entry is visited resulting in less precise Q-values. This can be counteracted by running the model a numerous amount of times, but I concluded that for this report, five was more than sufficient given the agent played ≈ 1000 rounds of Blackjack in total.

6 Conclusion and Future Work

This report presented a mathematical model for playing Blackjack that built on past work. This model incorporates techniques from Q-Learning and Neural Networks. The model also factors in expected payout and risk when determining actions.

References

[Bal+56] Roger R. Baldwin et al. "The Optimum Strategy in Blackjack". In: *Journal of the American Statistical Association* 51.275 (Sept. 1956), pp. 429–439. ISSN: 0162-1459, 1537-274X. DOI: 10.1080/01621459.1956.10501334.

[TW73] Edward Thorp and William Walden. "The Fundamental Theorem of Card Counting with Applications to Trente-et-Quarante and Baccarat". In: *Game Theory* 2 (1973), pp. 102–119.

[Sil92] Patrick Sileo. "The Evaluation of Blackjack Games Using a Combined Expectation and Risk Measure". In: *Gambling and Commercial Gaming: Essays in Business, Economics, Philosophy and Science* (1992).

[PS98] Andres Perez-uribe and Eduardo Sanchez. "Blackjack as a Test bed for Learning Strategies in Neural Networks". In: *In International Joint Conference on Neural Networks, IJCNN'98*. 1998, pp. 2022–2027.

[Dud15] Will Dudley. "The Probability of Winning at Blackjack". In: *The Personal Website of Vince Knight, Ph.D., Enumerative Combinatorics* (Dec. 2015).

[Wu18] Allen Wu. "Playing Blackjack with Deep Q-Learning". In: *Stanford University* (2018). URL: http://cs230.stanford.edu/files_winter_2018/projects/6940282.pdf.

[Grawn] Charles de Granville. "Applying Reinforcement Learning to Blackjack Using Q-Learning". In: *Oklahoma University* (Unknown). DOI: 10.1.1.495.4075. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.495.4075&rep=rep1&type=pdf.

Appendices

Symbol	Description
AB	Agent Bust
ABC	Agent Busting Cards
AT	Agent Total
DT	Dealer Total
A21	Agent Blackjack
D21	Dealer Blackjack
A21C	Agent Blackjack Cards
D21C	Dealer Blackjack Cards
UC	Unknown Cards
DLTC	Unknown Cards where Dealer Show + Card < AT

Table 1: Blackjack Probability Derivation Symbols

Iteration #	Games Played	Win Rate (%)	Loss Rate (%)	Draw Rate (%)
1	213	93.4272	6.5728	0
2	204	89.2157	9.8039	0.9804
3	197	92.8934	6.5990	0.5076
4	204	87.7451	10.2941	1.9608
5	199	84.4221	15.5779	0

Table 2: Model Results

Symbol	Bounds	Description
α	[0, 1]	The Q-Learning learning rate.
γ	[0, 1]	The Q-Learning discount factor.
β	[0, 1]	The NN weight update factor.
δ	[0, 1]	The NN bias update factor.
τ	[0, inf)	The softmax temperature.

Table 3: Model Parameters