Blackjack Reinforcement Learning: An Analysis of Q-Learning Techniques

Ian McAtee and Fletcher Wadsworth

EE5885: Deep Reinforcement Learning and Control
Professor Jiang
University of Wyoming

December 15, 2021

Abstract

This project examines the application of Q-learning techniques to train a reinforcement learning agent to play the game of blackjack. Specifically, the techniques of tabular Q-learning and deep Q-learning are implemented in a simulated blackjack environment. It is shown that while both methods achieve the learning of a blackjack policy, neither technique converged to an optimal policy. Both methods exhibit a blackjack win percentage of approximately 42%, failing to show significant improvement over standard heuristic blackjack policies. A discussion is presented that analyzes the possible reasoning behind these results, the suitability of Q-learning for this application, and examines the difficulties of using reinforcement learning techniques on the game of blackjack.

1 Introduction and Background

Blackjack is one of the most popular casino table games around the world. What differentiates blackjack from other games is the opportunity for strategic choices. Unlike games such as roulette, blackjack is structured so that a player's actions are correlated to their success, such that an optimal strategy can be employed to minimize that player's losses. Certain varieties of casino blackjack have known optimal strategies which are derived statistically and supported empirically [1]. However, applying Reinforcement Learning (RL) techniques to blackjack in an effort to generate an optimal policy experientially is a promising endeavor which aims to understand the underlying dynamics of competitive card games and improve a gambler's chances of winning.

Ultimately, blackjack is a comparatively simple game to which RL has been applied in a variety of implementations. The eventual target application is to the game of poker. Poker is a significantly more complex environment than blackjack; games have a larger state-action space comprised of more cards and available actions and other players are independent agents capable of their own decisions. With the immense number of permutations of a poker hand and the psychology of not entirely rational competitors influencing the trajectory of a hand, current poker strategies are only naively statistically supported and are largely intuitive. Thus, blackjack is the chosen environment used to build a foundation upon which the domain of poker can eventually become the target.

Here, agents are trained towards optimal blackjack strategy in a simulated environment using Q-Learning. Tabular Q-Learning and Deep Q-Learning methods are implemented and their resulting policies are compared in an effort to understand the advantages of each.

2 Methods and Results

2.1 Environment

To reduce the state and action space of the blackjack environment to a manageable level, the following rule-set is used in this implementation:

Blackjack Environment

- ⧫ Dealer and player each receive two cards from standard 52 card deck
  - ▷ One of the dealer's cards is face up and known to the player
- ⧫ Face cards (Jack, Queen, King) all valued at 10
- ⧫ Aces are valued at 11 or 1
- ⧫ Player has two available actions at each turn: **hit** and **stay**
- ⧫ At each turn, player selects one of the actions and is dealt another card or the hand terminates
- ⧫ After the player exceeds 21 (bust) or chooses to stay, the dealer resolves their hand
  - ▷ Dealer follows a fixed policy in which they must hit below 17
- ⧫ The winner is determined
  - ▷ If the player's hand exceeds the dealer's hand and the player has not busted, the player wins
  - ▷ If the player busts or the dealer's hand exceeds the player's without busting, the player loses
  - ▷ If neither the player or dealer bust and they have equal hand values, the hand is a draw (push)
- ⧫ A terminal state is reached and a reward is administered to the player based on the outcome of the hand

From this environment, the state, action, and reward spaces can be formalized as

$$S = \{sum,\ up\,card,\ usable\ ace\} \tag{1}$$
$$A = \{hit,\ stay\} \tag{2}$$
$$R = \{-wager,\ 0,\ wager\}, \tag{3}$$

where *sum* is the sum of the cards in the player's hand, *up card* is the visible dealer card, *usable ace* is whether the player hand contains an ace which can be reduced to 1, and *wager* is the bet amount won or lost. Note that the player agent has no control over the wager amount, making the reward a design parameter rather than a learned action.

2.2 Tabular Q-Learning

In Q-Learning, $Q(s, a)$ is known as the *action-value* function. $Q(s, a)$ outputs the value of a particular action in a particular state. The action-value function is formalized as

$$Q^{\pi}(s, a) = E\left[G_t | s_t = s, a_t = a\right] \tag{4}$$

where

$$G_t = r_t + \gamma\, r_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k\, r_{t+k}. \tag{5}$$

Thus, $Q(s, a)$ is the expected reward of executing an action a in the state s. In the tabular Q-Learning process, $Q(s, a)$ is not an analytical function, but rather a table which is populated and updated during learning and has a well defined output for each state-action pair. The values in this Q-table are learned using the update rule

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_a Q(s', a) - Q(s, a)\right] \tag{6}$$

where α is the learning rate, γ is the discount factor, and s' is the next state.

This update rule implies Q-Learning as an off-policy method; updates are made using a *greedy* policy by selecting the most rewarding action in the given next state, while the action a taken to reach the next state need not be chosen via a greedy policy. The policy used to select actions during exploration is known as the *behavioral* policy.

Q-learning can also be implemented offline. Offline RL denotes a learning process by which the static data set (which consist of $\{s, a, r, s'\}$ tuples) can be obtained before training and sampled during training. Such an offline implementation is utilized here with 3,000,000 simulated hands, and the training process is summarized:

Tabular Q-Learning

1. Dataset $D = \{s_i, a_i, r_i, s'_i\},\ i = 1, 2, \cdots N$ is obtained by rollout of a behavioral agent
    - ▷ Agent follows a random uniform policy, i.e. hit or stay with probability 0.5
2. Q-table initialized to 0
3. A single data tuple is sampled from D
4. Q-table entry is updated by Equation 6
5. Repeat 3,4 for all tuples in D

After the above learning procedure is complete, the learned policy is instantiated by a greedy query of the Q-table

$$\pi(s) = \max_a Q(s, a) \tag{7}$$

Once agent is trained on the data set, an evaluation of the agent's performance is performed over 200,000 hands. The win percentage of the trained agent is compared with that of one agent following a uniform random policy and one following a heuristic policy. The heuristic policy is to hit when the hand is valued below 17 and stay otherwise.

Examples of learned greedy policies are shown below, where the action with maximum value for each state is indicated. In these policy tables, red and blue denote *hit* and *stay* as the preferred action respectively:

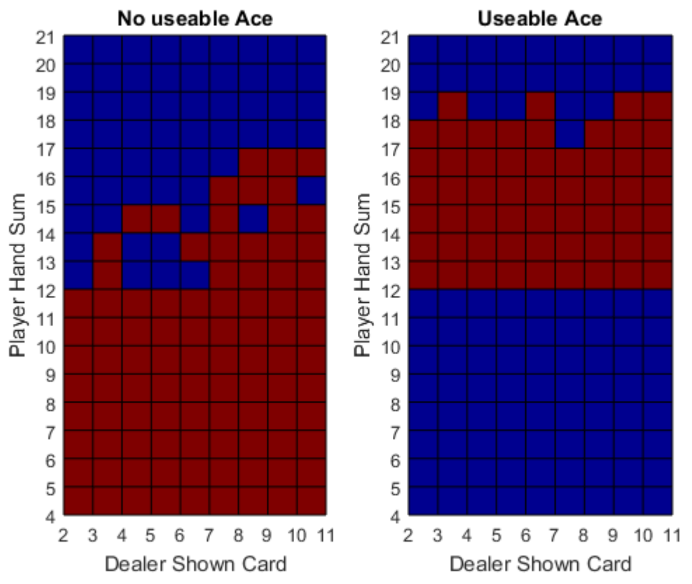

Figure 1: Greedy Policy with $\alpha = 0.01$ and $\gamma = 0.95$

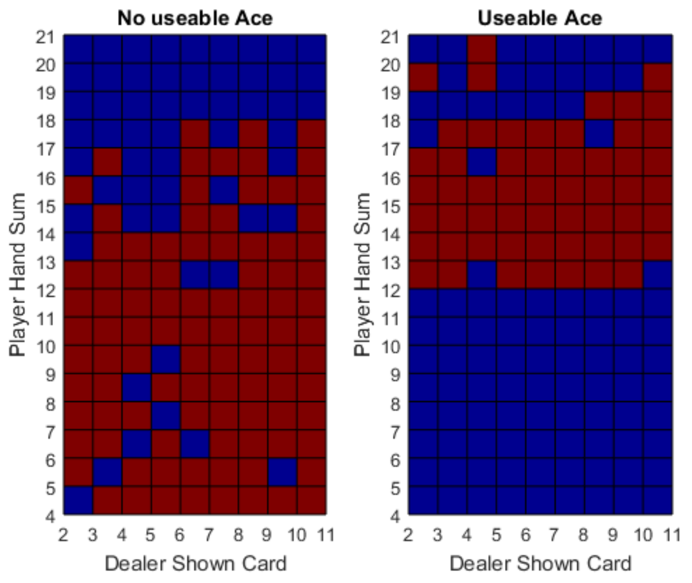

Figure 2: Greedy Policy with $\alpha = 0.3$ and $\gamma = 0.6$ (Red and Blue Denote Hit and Stay, Respectively)

The following tables show the performance evaluation response for the corresponding learned policies:

Trained Agent Win Rate	0.4184
Random Policy Win Rate	0.3042
Heuristic Policy Win Rate	0.4243

Table 1: Greedy Policy Performance with $\alpha = 0.01$ and $\gamma = 0.95$

Trained Agent Win Rate	0.4172
Random Policy Win Rate	0.3042
Heuristic Policy Win Rate	0.4223

Table 2: Greedy Policy Performance with $\alpha = 0.3$ and $\gamma = 0.6$

2.3 Deep Q-Learning

Having demonstrated the application of tabular Q-learning to the game of blackjack, deep Q-learning was posited as a possible method to improve results. Deep Q-learning is a neural network parameterized extension of the classical Q-learning algorithm. Recall that in tabular Q-learning, one must catalogue the Q-function, meaning that each action-state value pair must be tabulated, tracked, and updated throughout the course of learning. This is manageable given a limited and discrete action-state space, however, becomes increasingly intractable as this space expands. Deep Q-learning seeks to solve this intractability by using neural networks to approximate the Q-function of a state, rather than attempt to store and index an immense Q-table. The crux of deep Q-learning is the idea that a neural network can be trained as to accurately parameterize the Q-function such that one can obtain an estimate of the action-value for all actions given the state. This concept is shown below in figure 3.

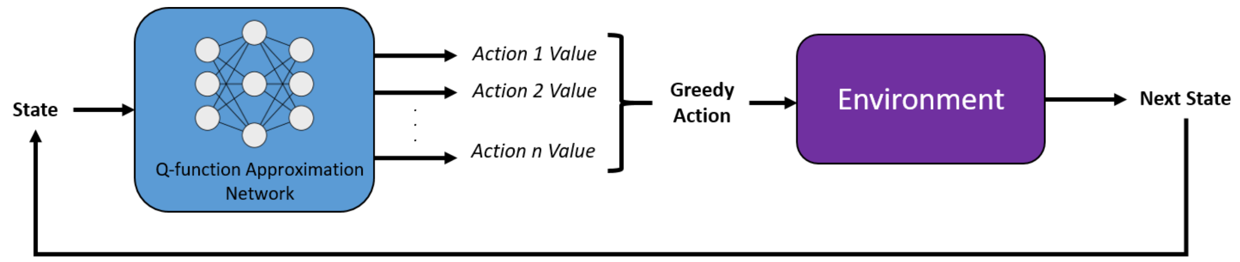

Figure 3: Overview of the Deep Q-Network Implementation

It should be noted that strictly speaking, deep reinforcement learning approaches are not essential for the solution of the blackjack problem. Recall that here, the state and action space, while large, have been shown to be tractable via tabular Q-learning methods. However, the goal of the application of deep Q-learning to the blackjack problem is to comprehend its implementation, compare the results against the tabular method, and to discuss its suitability as a possible solution to more complex card games such as poker.

2.3.1 Blackjack Environment Modifications

To apply a deep Q-learning approach to the blackjack problem, modifications had to be made to the environment as created for the tabular Q-learning method. In order to facilitate the straightforward application of the neural network parameterization, a prebuilt blackjack environment was imported from the *OpenAI Gym* library. This environment is of the same form as presented in section 2.1. However, the state space and reward function were modified slightly as shown:

$$S = \{sum,\ up\ card\} \tag{8}$$

$$A = \{hit,\ stay\} \tag{9}$$

$$R = \{-1, 0, 1.5\} \tag{10}$$

Here, one can see that the state space S was truncated as to not include the *usable ace* state. A result of the elimination of the *usable ace* is that the value of the ace was set at one for this instance of the blackjack environment. The state space was changed as to simplify its presentation to the associated Q-learning networks. Because the goal of the deep Q-learning implementation was to comprehend its operation, the networks were constructed from the ground-up. Thus, the state space reduction was performed purely to lessen the complexities of the neural network design process. It is recommended that future work in the application of deep Q-learning to blackjack include the *usable ace* state for the purposes of accuracy and completeness.

It can also be noticed that in equation 10, the reward function was also adjusted. This modification was made in attempt to facilitate better learning of the agent in the blackjack environment. The underlying belief is that because blackjack is inherently biased against the player, that is, the player is statistically predisposed to losing, the reward should be higher for a win as compared to a loss. However, this reward function still suffers from the issue of sparsity in time due to zero reward being yielded until the termination of the blackjack hand. However, a more robust discussion of the reward design process and its intrinsic difficulties in blackjack is given in section 3.2.

2.3.2 Deep Q-Learning Methods

The blackjack environment used allowed for the implementation of an *online* method of deep Q-learning. That is, the environment yields a training tuple $\{S, A, S', R\}$ at every time step. The pseudocode of the *online* deep Q-learning implementation is given below and a block-diagram overview is shown in figure 4.

Online Deep Q-Learning

1. Take ϵ-greedy action a_i to obtain the tuple sample $\{s_i, a_i, s'_i, r_i\}$ from environment
2. Compute TD target as: $y_i \leftarrow r(s_i, a_i) + \gamma max_{a'} Q(s'_i, a'_i; w')$
    - $Q(s'_i, a'_i; w')$ is computed using **target network**
3. Update the **Q-function approximation network** weights
    - $w \leftarrow w - \alpha \sum_i \Delta_w Q(s_i, a_i; w)(Q(s_i, a_i; w) - y_i)$
    - $Q(s_i, a_i; w)$ is computed using **Q-function approximation network**
4. Update **target network** w' with **Q-function approximation network** w every M steps
5. Repeat 1-4 for all training tuples

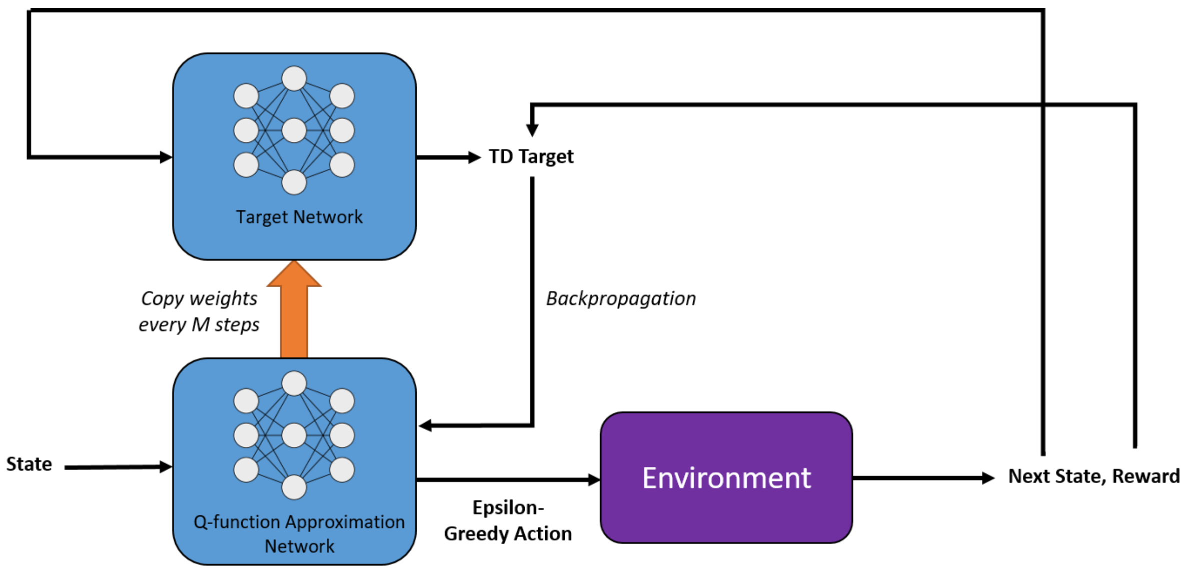

Figure 4: Overview of Deep Q-learning Network Training

As one can see, this implementation involves the creation of duel neural networks. The *Q-function approximation network* is the neural network parameterization of the Q-function. It is input with the state space and provides an estimation of the action value of each of the possible actions in the action space. Here, an ϵ-greedy policy is used to select the action taken by the agent in the environment. That is, the agent will with a $1 - \epsilon$ probability, take the greediest action. This greedy action is the one associated with the maximal action-value estimation. It is this Q-function approximation network that one wishes to optimize through learning such that it is an accurate representation of the Q-function and thus can yield a satisfactory policy.

However, the training of this network can prove difficult due to challenges inherent in Q-learning. These challenges are well-known and studied and include issues of nonstationarity and sample correlation. If one examines equation 6, it can be noticed that the *temporal difference* (TD) target $r + max_a Q(s', a)$ is not stationary. That is, the target continuously changes upon each iteration. Recall that in normal deep learning neural networks, the target is stationary and thus the training is stable. Consequently, this non-stationary target in deep Q-learning means that training can exhibit highly divergent behavior. The solution to this is to introduce a separate but identically architectured network to calculate the target value. This target network exhibits set parameters that are only updated every M training steps as a cloning of the Q-function approximation network parameters. The rational is that by fixing the target values, the training becomes pseudo-stationary within the M-step training window.

The issue of sample correlation arises due to the online nature of the deep Q-learning algorithm. The sample tuples are generated from the environment sequentially, meaning that in general, they are highly correlated. To account for this, the Q-learning algorithm usually implements experience replay. In experience replay, generated samples are stored in a replay buffer and batched samples are randomly drawn from the buffer to perform the learning. This random drawing of the samples provides for the decorrelation of the training samples. However, the highly stochastic nature of blackjack means that sequence of state transitions does not happen in a linear and organized way. Therefore, the assumption is made that the samples generated by the blackjack environment are not particularly correlated and thus, the implementation of experience replay is not necessary. A deeper discussion of this assumption is presented is section 3.1.

2.3.3 Deep Q-Learning Results

The deep Q-learning method described in section 2.3.2 was applied to the modified blackjack environment. Through experimentation, a simple two hidden layer fully connected neural network was found to be sufficient for blackjack Q-function approximation. The specifics of the network architecture used for both the Q-function approximation network and the target network is given in table 3.

Layer	**Number of Nodes**	**Activation Function**
Input	2	Identity
Hidden 1	32	RELU
Hidden 2	16	RELU
Output	2	SoftMax

Table 3: Neural Network Architecture of Q-Function Approximation and Target Networks

Training of the Q-function approximation network was accomplished by implementing deep Q-learning with 2500 samples generated by the blackjack environment in an online paradigm. The following parameters in table 4 were used to implement the deep Q-learning:

Parameter	**Value**
Exploration Rate ϵ	0.1
Discount Factor γ	0.9
Learning Rate α	0.001
Target Network Update Interval M	100 Steps

Table 4: Deep Q-Learning Parameters

The 2500 samples were organized into 50 episodes, each with 50 samples, in an attempt to analyze how the average reward gained by the agent changed over time. Figure 5 plots this average reward versus the number of episodes.

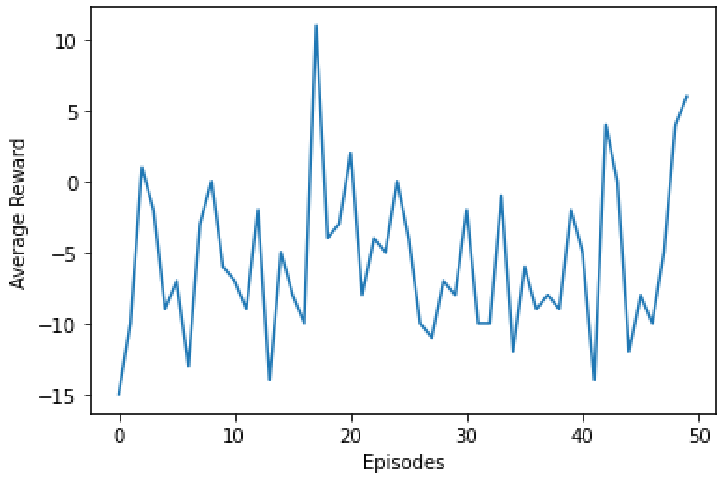

Figure 5: Average Reward Over Training Episodes

An examination of figure 5 shows that the deep Q-learning was unable to produce an upward trend in the average rewards. Instead, it is observed that the average reward steadily fluctuates around zero, with the average reward of most episodes remaining negative. This demonstrates the difficulties inherent to blackjack. The fact that the game is biased against the player will make it difficult for any reinforcement learning agent to obtain a steady increase in average rewards and terminate with a policy that will consistently yield positive rewards.

To analyze the resultant blackjack policy achieved by deep Q-learning, the trained Q-function approximation network was used to evaluate the Q-function over the entire state space. The maximum action-value was then used to determine the action the policy would implement in each particular state. This resulted in the policy shown in figure 6.

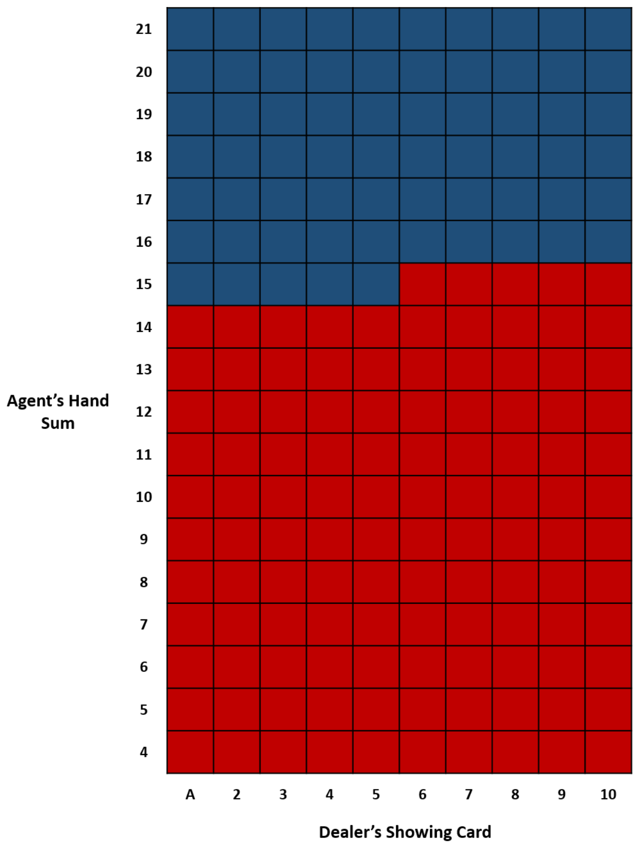

Figure 6: Blackjack Policy Learned by Deep Q-Learning (Red and Blue Denote Hit and Stay, Respectively)

Figure 6 shows that the policy found by deep Q-learning is much more uniform than that achieved by tabular Q-learning. This is likely the result of the neural network approximation of the Q-function. Unlike tabular methods, the deep approach means that the neural network can generalize, even if a particular state hasn't been experienced often. Therefore, one would expect a more uniform policy. However, this does not necessarily mean the policy is superior to the tabular policy. This generalization may mean that subtleties in the state-transition statistics have been lost that may be relevant in defining a truly optimal policy. In fact, the policy produced by deep learning is strikingly similar to that of the heuristic *hit under 17* policy, which is known to not be the optimal policy. Therefore, it is unsurprising that upon testing, the deep Q-learning policy and the heuristic policy exhibit similar win percentages, as shown in table 5.

Policy	**Win Percentage**
Heuristic	0.428
Deep Q-Learning	0.425

Table 5: Win Percentages of Heuristic and Deep Q-Learning Policies as Evaluated Over 1000 Blackjack Rounds

3 Discussion

3.1 State Transition Stochasticity

The win percentage of the trained agents demonstrate that the agents have learned a policy sufficient to outperform a random policy. While this is promising, the inability of the agents to significantly outperform the heuristic policy is worthy of exploration.

Consider the following region of the state space in which the player has no usable aces:

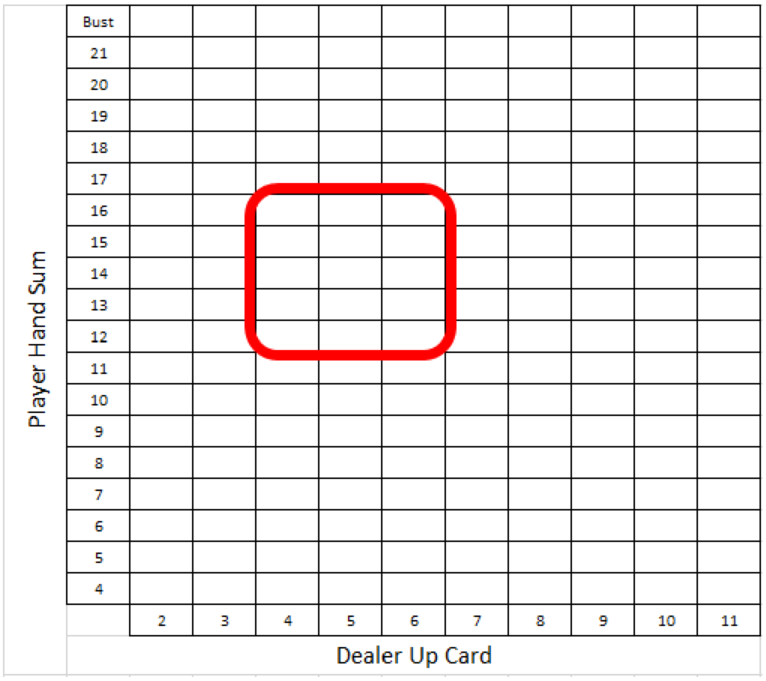

Figure 7: State Space Region of Interest

This region represents a departure from the heuristic policy which can be reasonably analyzed. Since all face cards are valued at 10, the plurality of the deck is valued at 10. Assuming multiple decks are combined (commonplace in casino blackjack), regardless of the cards dealt to the dealer and players, there is an approximate 30.8% chance that the dealer has a 10 card hidden, and the same chance of the player or the dealer drawing a 10 card. This means that in this region, both the player and dealer are more likely to bust than not. However, the dealer is beholden to the policy placed on them, and must hit for hands less than 17. From this it can be surmised that despite a sub-optimal player hand (not valued high enough for a reasonable chance of beating another hand), the dealer is likely to bust, and thus the player should stay.

However, this is a rather subtle advantage for the player. Since the player does not know the entire hand of the dealer, and cards are drawn randomly, the outcome of this particular premise has high variance. To testify to this point, consider the following state transition:

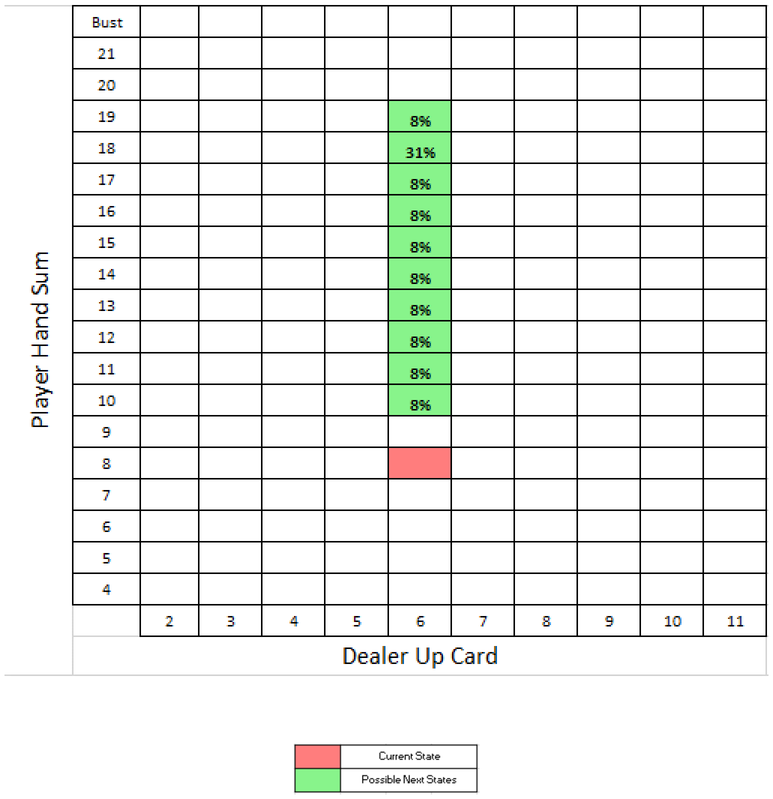

Figure 8: State Transition Probability

In any given state, the number of states to which can be transitioned by choosing to hit is at most 10. Furthermore, the distribution of the possible next state has fairly high variance. These examples illustrate the profound stochasticity of the state transitions inherent in card games such as blackjack. As a consequence, an enormous amount of experiential data is needed for the agent to learn the marginal advantage that one action may possess over another in a given state which does not appear naturally or obviously.

3.2 Reward Design

The reward schedule which comes intuitively from the structure of blackjack is to tender a reward when the hand has been resolved and a winner decided and tender no reward for agent actions during the episode. This is a natural reward scheme, since in an actual game of blackjack, a player wins no money and is not compensated or penalized for taking actions per se. This leads to some issues with regard to the Q-table and network parameter updates with regard to tabular and deep Q-Learning respectively. Consider Equation 6; with the current reward schedule, every $\{s, a, r, s'\}$ tuple collected has the form of $\{s, a, 0, s'\}$ or $\{s, a, r, s_{terminal}\}$. Thus every update contains a reward term or a discounted next state greedy action-value, but not both. Such an environment can be characterized as a sparsely rewarded environment. Sparse rewards make updating Q-table values difficult, particularly at the beginning of learning, as the discounted next state action-value may not be populated or contain meaningful information. Similarly for the parameter update in deep Q-Learning, initial learning with sparse rewards results in poor initial learning and is subject to the forces of random weight initialization and uncorrelated data.

A simple and obvious strategy to overcome these sparse reward difficulties is more training time and a larger training data set. This is a through line to card game problems in RL as has been discussed; the narrow statistical advantages throughout the state space and sparse reward schedule benefit largely from expansive training. However, a redesign of the reward schema could be a viable tactic to combat the need for ever-larger training time. While the implementations here utilized the intuitive reward scheme discussed, alternatives were considered. In particular, different scaling of the wager amount showed no notable effect on the learned policy. Other potential renovations to the reward schedule include different ratios of reward for different outcomes (i.e. change the ratio of punishing a losing hand to rewarding a winning one), a dynamic reward schedule where the reward amount would either decay or increase as training progresses, and adding rewards to actions taken regardless of the outcome of the episode. It can also be noted that a different RL technique entirely could be effectively utilized to attenuate this problem. Monte-Carlo Learning is a possible alternative to Q-Learning, as it learns directly from sampled trajectories. Revising of the reward schema and exploration of other RL techniques are future endeavors to further develop this problem.

4 Conclusion and Future Work

This paper analyzed blackjack as an application for reinforcement learning. In particular, tabular and deep Q-Learning were used to train a blackjack agent in hopes of reaching an optimal policy, wherein the trained agent could minimize its losses in a series of simulated games and perform better than an agent following a more heuristic but less robust policy. In these implementations, the trained agents performed about on par with the heuristic policy and failed to achieve an optimal policy. Several observations have been made about possible reasons for the underwhelming trained policy, including the profoundly stochastic nature of card games, the need for a considerable amount of training time and data, and the difficulty of designing a proper reward schedule.

In the future, fine tuning of the learning hyperparameters and network structure may help to improve the agent's policies, particularly in the deep Q-Learning domain. Exploration of different reward frameworks and perhaps different RL algorithms all together may be more suited to deriving an optimal policy in the context of blackjack and other card games. After these problems are addressed, the expansion of the state and action space to equate the environment to proper casino blackjack and eventual tailoring to more complex card games such as poker is endeavored.

5 Appendix

5.1 Tabular Q-Learning Listings

- Generate Data Script

```
%Script generates {s,a,r,s'} data tuples for Q-learning of
%2 player blackjack environment using uniform random
   [+]behavioral policy
clear all
tic
%ACTION: [1,2] == ['stay','hit']
numEpisodes = 3000;
handsPerEpisode = 1000;

ante = 10; %reward amount
numPlayers = 2; %Agent and dealer
numCards = 2; %Blackjack initial hand = 2 cards
numDecks = 4;
aceHigh = 1;

encode_state = @(hand,dCard,ace) (hand-1) + 18*(dCard-2)+
   [+]180*(ace);

SARS = int16([]);
SARS_total = int16([]);

for i = 1:numEpisodes
    rng('shuffle')
    SARS=int16([]);
    parfor ii = 1:handsPerEpisode
        handOver = 0;
        while ~handOver
            %------Params and storage arrays------
            state = [];
            state_fcn = [];
            action = [];
            reward = [];

            %-----------Simulate Hand-----------
            %Deal cards to numCards x numPlayers matrix
            [dealtCards,deck] = dealCards(numPlayers,numCards,
               [+]numDecks,aceHigh);
            %Split cards into player and dealer hands
            pHand = dealtCards(1,:);  dHand = dealtCards(2,:);
            %Evaluate player hand
            [pHand,pVal,pBust,pUseAce] = HandEval(pHand);

            %Select action using random policy
            while (~pBust)&&(~handOver)
                state = [state, [pVal;dHand(2);pUseAce]];
                state_fcn = [state_fcn, encode_state(pVal,
                   [+]dHand(2),pUseAce)];
                reward = [reward 0];
                actSelect = rand;
                if actSelect < 0.5
                    action = [action, 2];
                    [pHand,deck] = PlayerHit(pHand,deck);
                    [pHand,pVal,pBust,pUseAce] = HandEval(
                       [+]pHand);
                else
                    action = [action, 1];
                    handOver = 1;
                    break
                end
            end
            %Resolve dealer hand
            [dBust,dBJ,dVal,deck] = DealerPolicy(dHand,deck);
                %Determine winner
                if (pBust) || ((dVal > pVal) && (~dBust))
                    reward = [reward, -ante];
                    %Lose state encoded by 0
                    state_fcn = [state_fcn, 0];
                elseif (pVal > dVal) || dBust
                    reward = [reward, ante/5];
                    %Win state encoded by 2
                    state_fcn = [state_fcn, 2];
                else
                    reward = [reward, 0];
                    %Draw state encoded by 1
                    state_fcn = [state_fcn, 1];
                end
                reward(1) = [];
                for p =1:length(action)
                    SARS = [SARS; [state_fcn(p), action(p),
                       [+]reward(p),state_fcn(p+1)]];
                end
        end
    end
    SARS_total = [SARS_total;SARS];
end
```

```
toc
save('SARS_data.mat','SARS_total','-mat')
```

- Deal Cards Function

```
function [dealtCards,deck] = dealCards(numHands, cardsPerHand,
   [+] numDecks,aceHigh)
%function deals cards from specified number of decks for input
   [+] number of
%cards to input number of players
    deck = [];
    one = ones(1,4); %1x4 vector of ones
    cards = zeros(cardsPerHand,numHands);
    if aceHigh
        c_set = [2 3 4 5 6 7 8 9 10 10 10 10 11];
    else
        c_set = [1 2 3 4 5 6 7 8 9 10 10 10 10];
    end
    %Repeat for each 52 card deck desired in complete shoe
    for d = 1:numDecks
        %Repeat for each number 1-10 plus 3 face cards (Ace
           [+]encoded as 1)
        for c = 1:4
           %Append four cards of given number to deck, one for
              [+] each suit
           deck = [deck; c_set];
        end
    end
    %make deck into a row vector
    deck_vec = reshape(deck,1,[]);
    %Shuffle deck
    deck_vec = deck_vec(randperm(length(deck_vec)));
    %Repeat for desired number of cards per hand
    for cc = 1:cardsPerHand
        %Repeat for number of players
        for p = 1:numHands
            %Deal final card in deck
            cards(p,cc) = deck_vec(end);
            %remove dealt card from deck
            deck_vec(end) = [];
        end
    end
    dealtCards = cards;
    deck = deck_vec;
```

```
end
```

- Player Hand Evaluation Function

```
function [newHand, value , bust , useableAce] = HandEval(hand)
%function evaluates current player hand
%
%Outputs:
%
%   newHand: hand with face cards and aces resolved
%   value: sum of current hand
%   bust: flag indicating if hand value exceeds 21
%   useableAce: flag indicating whether hand has ace of value
    [+]11

    %Determine aces in hand
    heldAces = (hand==11);
    numAces = sum(heldAces);

    %If no aces in hand
    if numAces == 0
        %Find hand value
        value = sum(hand);
        useableAce = 0;
        if value <= 21
            bust = 0;
        else
            bust = 1;
        end

    %If at least one Ace in hand
    else
        %Find hand value
        value = sum(hand);
        %if not bust
        if value <= 21
            useableAce = 1;
            bust = 0;
        else
            aceCount = numAces;
            value_temp = value;
            while value_temp > 21 && aceCount > 0
                value_temp = value_temp - 10;
                aceCount = aceCount - 1;
            end
            if value_temp > 21
                useableAce = 0;
                bust = 1;
            else
                a = find(hand==11);
                k = 1;
                while value > 21
                    hand(a(k)) = 1;
                    value = sum(hand);
                    k = k+1;
                end
                bust = 0;
                if isempty(find(hand==11))
                    useableAce = 0;
                else
                    useableAce = 1;
                end
            end
        end
    end
    newHand = hand;
end
```

- Player Hit Function

```
function [newHand,newDeck] = PlayerHit(hand,deck)
%Function administers card to input hand from input deck
    card = deck(end);
    deck(end) = [];
    newHand = [hand, card];
    newDeck = deck;
end
```

- Dealer Policy Function

```
function [bust,blackjack,handValue,newDeck] = DealerPolicy(
   [+]hand, deck)



    %Obtain deck size (13k x 4)
    deckSize = size(deck);

    %Initialize flags and counter
    blackjack = 0;
    bust = 0; % Control flag to indicate hand score above 21
    hit = 1; % Control flag to indicate hand score below 17
    cardCount = 2; % Counter of hand size
    Hand = hand; % Initial dealer hand vector without suits

    %Repeat process as long as dealer has not busted and hand
       [+]value is
    %below 17
    while ~bust && hit
        %Hand(Hand > 10) = 10; % Replace face cards with value
           [+] of 10
        heldAces = (Hand == 1); % Indices of aces in dealer
           [+]hand
        aces = sum(heldAces); % Number of aces in dealer hand
        %Evaluate for no aces in hand
        if aces == 0
            handValue = sum(Hand); % Value of hand

            if handValue > 21 % Dealer bust
                bust = 1;
            elseif handValue > 16 % Dealer must stay above 17
                bust = 0;
                hit = 0;
            else
                card = deck(end);
                deck(end) = [];
                %Append drawn card to hand vector
                Hand = [Hand, card];
            end
        %At least one Ace in dealer hand
        else
            Hand(heldAces) = 11;
            handValue = sum(Hand);
            %Reduce soft hands, use ace as 1 instead of 11
            while handValue > 21 && aces > 0
                handValue = handValue - 10;
                aces = aces - 1;
            end
            if handValue > 21 %Dealer bust
                bust = 1;
            elseif handValue > 16 %Dealer must stay above 17
                bust = 0;
                hit = 0;
            else
                card = deck(end);
                deck(end) = [];
                %Append drawn card to hand vector
                Hand = [Hand, card];
            end
        end
    end
    newDeck = deck;
    handValue = sum(Hand);
end
```

- State Feature Decoding Function

```
function [ state ] = StateFunction2Table( enc_state )
%Function decodes encoded state value (scalar) into individual
%state features (1x3 vector)
%
%   encoded_state = (sum - 1) + 18(dealerCard -2) + 180(
   [+]useableAce)
%
%   Output: state = [sum,dealerCard,useableAce]
    orig_enc_state = enc_state;
    if enc_state < 183
        ace = 0;
    else
        ace = 1;
        enc_state = enc_state - 180;
    end

    d = mod(enc_state,18)+1;

    if d >= 4
        sum = d;
    else
        sum = 18+d;
    end
    dealerCard = (orig_enc_state - 180*ace - (sum-1))/18 + 2;
    state = [sum,dealerCard,ace];
end
```

- Q-Learning Script

```
%Q-learning script
clear all
%Load data
import = load('SARS_data.mat');
SARS = import.SARS_total;
SARS = single(SARS);
```

```
%State feature vectors
sums = 4:21;
shownCards = 2:11;
aces = 1:2;
actions = 1:2;
%Hyperparameters
alpha = 0.3;
gamma = 0.6;
%Check imported data array
[data_samples,sample_size] = size(SARS)
if sample_size ~= 4
    error(”Data points in SARS not correct dimensions”)
end

%------Parse Data into state, action, reward, and next state
   [+]vectors------
%Decode states from integer encoding to state indices for Q
   [+]table
s = zeros(data_samples,3);
sp = zeros(data_samples,3);
for n = 1:data_samples
    s(n,:) = StateFunction2Table(SARS(n,1));
    sp(n,:) = StateFunction2Table(SARS(n,4));
end
%Slice action and reward vectors
a = SARS(:,2); r = SARS(:,3);
%Change encoding of useable ace state component for
%use as index
s(:,3) = s(:,3)+1;
sp(:,3) = sp(:,3)+1;
%Initialize Q-table and state visit counter
Q_table = zeros(sums(end),shownCards(end),aces(end),actions(
   [+]end));
state_visit = zeros(sums(end),shownCards(end),aces(end));
%Update Q-table
for k = 1:data_samples
    maxQ = max(Q_table(sp(k,1),sp(k,2),sp(k,3),:));
    Q_table(s(k,1),s(k,2),s(k,3),a(k)) = Q_table(s(k,1),s(k,2)
       [+],s(k,3),a(k)) ...
        + alpha*(r(k) + gamma*maxQ - Q_table(s(k,1),s(k,2),s(k
           [+],3),a(k)));
    state_visit(s(k,1),s(k,2),s(k,3)) = state_visit(s(k,1),s(k
       [+],2),s(k,3))+1;
end
```

```
%Save trained Q-table
save('TrainedQTable.mat','Q_table','-mat')

%------------------------Plotting------------------------------
Q_plot = zeros(sums(end),shownCards(end),aces(end));

for ii = 4:sums(end)
    for jj = 2:shownCards(end)
        for kk = aces
            if Q_table(ii,jj,kk,2) > Q_table(ii,jj,kk,1)
                Q_plot(ii,jj,kk) = 2;
            else
                Q_plot(ii,jj,kk) = -2;
            end
        end
    end
end

figure(1)
subplot(1,2,1)
surf(Q_plot(:,:,1))
axis([2 11 4 21])
xticks(2:11)
yticks(4:21)
colormap('jet')
title('No useable Ace')
view(0,90)
ylabel('Player Hand Sum')
xlabel('Dealer Shown Card')


subplot(1,2,2)
surf(Q_plot(:,:,2))
axis([2 11 4 21])
xticks(2:11)
yticks(4:21)
colormap('jet')
title('Useable Ace')
view(0,90)
ylabel('Player Hand Sum')
xlabel('Dealer Shown Card')

figure(2)
subplot(1,2,1)
surf(Q_table(:,:,1,1))
```

```
axis([2 11 4 21])
xticks(2:11)
yticks(4:21)
colormap('jet')
title('Stay Value')
view(0,90)
ylabel('Player Hand Sum')
xlabel('Dealer Shown Card')


subplot(1,2,2)
surf(Q_table(:,:,1,2))
axis([2 11 4 21])
xticks(2:11)
yticks(4:21)
colormap('jet')
title('Hit Value')
view(0,90)
ylabel('Player Hand Sum')
xlabel('Dealer Shown Card')

figure(3)
subplot(1,2,1)
surf(state_visit(:,:,1))
axis([2 11 4 21])
xticks(2:11)
yticks(4:21)
colormap('jet')
title('Ace')
view(0,90)
ylabel('Player Hand Sum')
xlabel('Dealer Shown Card')


subplot(1,2,2)
surf(state_visit(:,:,2))
axis([2 11 4 21])
xticks(2:11)
yticks(4:21)
colormap('jet')
title('No Ace')
view(0,90)
ylabel('Player Hand Sum')
xlabel('Dealer Shown Card')
```

- Trained Agent Comparison Simulation Script

```
%Script to run games to test agent performance
rng('shuffle')
import = load('TrainedQTable.mat', '-mat');
Q = import.Q_table;

numPlayers = 2; %Agent and dealer
numCards = 2; %Blackjack initial hand = 2 cards
numDecks = 4; %number of decks in shoe
aceHigh = 1; %ace high encoding
numGames = 200000; %number of simulation games
%Initialize win counters
trainedPolicyWin = zeros(1,numGames);
randomPolicyWin = zeros(1,numGames);
fixedPolicyWin = zeros(1,numGames);
for i = 1:numGames
    handOver = 0;
    win = [];
    %Deal cards to numCards x numPlayers matrix
    [dealtCards,deck] = dealCards(numPlayers,numCards,numDecks
       [+],aceHigh);
    %Split cards into player and dealer hands
    pHand = dealtCards(1,:);  dHand = dealtCards(2,:);
    shownCard = dHand(2);
    %Evaluate player hand (triplicate, one for each player)
    [pHand,pVal,pBust,pUseAce] = HandEval(pHand);
    %Store evaluations in array to evaluate each player
       [+]separately by idx
    Hand = [pHand; pHand; pHand];
    Val = [pVal; pVal; pVal];
    Bust = [pBust; pBust; pBust];
    Ace = [pUseAce; pUseAce; pUseAce];
    Ace = Ace + 1;
    deck_reuse = deck;
    for plyr = 1:3
        hand = Hand(plyr,:);
        handOver = 0;
        deck = deck_reuse;
        while ~Bust(plyr) && ~handOver
            act = ChooseAction(hand,Val(plyr),shownCard,Ace(
               [+]plyr),plyr,Q);
            if act == 2
                [hand,deck] = PlayerHit(hand,deck);
                [hand,Val(plyr),Bust(plyr),Ace(plyr)] =
                    [+]HandEval(hand);
                else
                    handOver = 1;
                end
            end
        end
        [dBust,dBJ,dVal,deck] = DealerPolicy(dHand,deck);
        for plyr = 1:3
            %Determine winner
            if (Bust(plyr)) || ((dVal > Val(plyr)) && (~dBust))
                win = [win 0];

            elseif (Val(plyr) > dVal) || dBust
                win = [win 1];
            else
                win = [win 0];
            end
        end
        trainedPolicyWin(i) = win(1);
        randomPolicyWin(i) = win(2);
        fixedPolicyWin(i) = win(3);

end
%Calculate win % for each agent
tWinRate = sum(trainedPolicyWin)/length(trainedPolicyWin)
rWinRate = sum(randomPolicyWin)/length(randomPolicyWin)
fWinRate = sum(fixedPolicyWin)/length(fixedPolicyWin)
```

- Simulation Select Action Function

```
function [ action ] = ChooseAction( hand,handVal,shownCard,
   [+]uAce,agent_type,Q_table )
%function chooses agent actions (hit,stay) for AgentAnalytics.
   [+]m
    %Trained agent
    if agent_type == 1
        action = max(Q_table(handVal,shownCard,uAce,:));
    %Random agent
    elseif agent_type == 2
        actRand = rand;
        if actRand < 0.5
            action = 1;
        else
            action = 2; %action = 2;
        end
        %Fixed policy agent
    else
        if handVal < 17
            action = 2;
        else
            action = 1; %action = 2;
        end
    end
end
```

5.2 Deep Q-Learning Listing

```
#DEEP Q-LEARNING FOR BLACKJACK
#AUTHOR: Ian McAtee
#DATE: 12/06/2021
#CLASS: EE5885 Deep-Reinforcement Learning
#DESCRIPTION: Script to perform the deep Q-learning for an angent
   [+]to play blackjack

#DQN AGENT CLASS

import random
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from tensorflow.keras.optimizers import Adam

class DQNAgent():
    def __init__(self, env, epsilon=1.0, alpha=0.5, gamma=0.9):

        self.action_size = 2 #Size of action space
        self.state_size = env.observation_space  #Size of state
           [+]space
        self.epsilon = epsilon                   #Exploartion
           [+]factor
        self.alpha = alpha                       #Learning Rate
        self.gamma = gamma                       #Discount Factor
        self.model = self.QNetwork()             #QNetwork
           [+]Approximation
        self.target = self.QNetwork()            #Target Network

    #Construct Q-function approximation network
    def QNetwork(self):
        model = Sequential()
        model.add(Dense(32, input_shape = (2,), kernel_initializer
           [+]='random_uniform', activation='relu'))
        model.add(Dense(16, activation='relu'))
        model.add(Dense(self.action_size, activation='softmax'))
        model.compile(loss='mse', optimizer=Adam(learning_rate=
           [+]self.alpha))

        return model

    #Function to choose action based on epsilon greedy policy
    def chooseAction(self, state):

        #Random Action
        if random.random() < self.epsilon:
            action = random.choice([0,1])

        #Greedy Action
        else:
            action_value = self.model.predict(state)
            action = np.argmax(action_value[0])
        return action

    #Function to do the deep Q-learning
    def DQlearning(self, state, action, reward, next_state, done,
       [+]steps):

        #Set the target

        #If done with hand set target to the reward (because there
           [+] is no next state)
        if done:
            y = reward

        #If not done with hand set target as in Q-learning
           [+]equation, uses target network
        else:
            y = reward + self.gamma * np.amax(self.target.predict(
               [+]next_state)[0])

        #Use Q-network approximation to predict the Q-function for
           [+] current state
        Q_sa = self.model.predict(state)

        #Set the current action's index in the predicted Q-
           [+]function to the target
```

```
            Q_sa[0][action] = y

            #Train model
            self.model.fit(state, Q_sa, epochs=1, verbose=0)

            #M is number of steps before updating target network
            M = 200;

            if steps%M == 0:
                self.updateTarget() #Update target netwrok

    #Function to update the target network
    def updateTarget(self):
        self.target.set_weights(self.model.get_weights())

    #Function to evaluate trained QNetwork given a state sample
    def evaluateModel(self,state):
        pred = max(self.model.predict(state))
        maxQ = max(pred)
        if pred[0] == maxQ:
            a = 0
        else:
            a = 1
        return a


#DEEP Q-LEARNING TRAINING

import gym
import matplotlib.pyplot as plt

env = gym.make('Blackjack-v0')
env.seed(0)

#Number of hands to play in each episode
numSamps = 50

#Number of episodes to play
numEpisodes = 50

#Create deep Q learning Agent
agent = DQNAgent(env=env, epsilon=0.1, alpha=0.001, gamma=0.9)

#Define the average reward
averageReward = []
```

```

#Initialize a step counter
step = 1

#Setup the enivronment and reshape the state space for neural
    [+]network
state = env.reset()
state = np.reshape(state[0:2], [1,2])

#Perform the Deep Q-learning
for episode in range(numEpisodes): #Loop Episodes

    print('Training: Episode ',episode)

    totalReward = 0

    for samp in range(numSamps): #Loop samples

        #Take action based on epsilon-greedy policy
        action = agent.chooseAction(state)
        #Apply action to environment to get next state, reward,
            [+]and is_done signal
        next_state, reward, done, _ = env.step(action)
        #Reshape state space to use with neural network
        next_state = np.reshape(next_state[0:2], [1,2])
        #Tally the reward
        totalReward += reward
        #Perform the deep Q-learning
        agent.DQlearning(state, action, reward, next_state, done,
            [+]step)
        #Go to next state
        state = next_state
        state = np.reshape(state[0:2], [1,2])
        #If blackjack hand done, reset the environment
        if done:
            state = env.reset() # Environment deals new cards to
                [+]player and dealer
            state = np.reshape(state[0:2], [1,2])

        step += 1

    averageReward.append(totalReward)


```

```
#Plot the average rewards over the episodes
plt.plot(averageReward)
plt.xlabel('Episodes')
plt.ylabel('Average Reward')
plt.show()

#DETERMINE POLICY

#Get policy table, by evaluating Q-funtion approximation network
possHandSum = range(4, 22)
possDealUpCard = range(1, 11)
policyTable = [[str(-1) for i in possDealUpCard] for j in
    [+]possHandSum]
for i in possHandSum:
    for j in possDealUpCard:
        policyTable[i-4][j-1] = str(agent.evaluateModel([[i,j]]))

#Display the Policy
print('                              POLICY ')
print('')
print('Agent Hand |                  Dealer Up Card')
print('   Sum     |   A    2    3    4    5    6    7    8    9
    [+]   10')
print('
    [+]------------------------------------------------------------------------------
    [+]')
for i in range(4,10):
    print('    ',i,'      |',policyTable[i-4])
for i in range(10,22):
    print('   ',i,'      |',policyTable[i-4])

#TEST THE FOUND POLICY AGAINST THE HEURISTIC POLICY

#Resetl the environment
env.seed(0)
state = env.reset()


#Test Heuristic Hit Under 17 Policy
numHands = 1000
winsH = 0

for i in range(numHands):
    state = env.reset()
    done = False
    while not done:

        #Hit or stay based on if you hold less than 17
        if state[0] < 17:
            action = 1
        else:
            action = 0

        #Get the next state
        next_state, reward, done, _ = env.step(action)

        state = next_state

        if reward == 1.0:
            winsH += 1

#Test Trained Q-Learning Policy
numHands = 1000
winsQ = 0

for i in range(numHands):
    state = env.reset()
    done = False
    while not done:

        #Evaluate the q-function approximation network
        action = agent.evaluateModel([[state[0],state[1]]])

        #Get the next state
        next_state, reward, done, _ = env.step(action)

        state = next_state

        if reward == 1.0:
            winsQ += 1

#Display the win results
print('Heuristic Wins: ',winsH)
print('Qlearning Wins: ',winsQ)
```

References

[1] E. Thorpe, "A Favorable Strategy for Twenty-One. *Proc Natl Acad Sci USA*, vol. 47, 1, 1961