BLACKJACK PLAYING AGENT WITH MACHINE LEARNING

by

EGEMEN ÇİMEN

Engineering Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

2019

BLACKJACK PLAYING AGENT WITH MACHINE LEARNING

APPROVED BY:

Assist. Prof. Dr. Dionysis Goularas……
(Supervisor)

Assist. Prof. Dr. Tacha Serif ……

Assist. Prof. Dr. Esin Onbaşıoğlu ……

DATE OF APPROVAL: 14/01/2019

ACKNOWLEDGMENTS

I would like to thank my advisor Assist. Prof. Dr. Dionysis Goularas for his guidance and support throughout my project. I also would like to thank my mother, and my maternal grandparents for their continued support.

ABSTRACT

BLACKJACK PLAYING AGENT WITH MACHINE LEARNING

Blackjack or twenty-one is a card game that gives the player a chance to have an advantage over the dealer. The objective of the game is having a higher total of card values than the dealer's, while also having the total to be smaller or equal to the value 21. The player can use a "basic strategy" to even the odds with the dealer. When the player also takes the distribution of cards played in the prior hands into consideration, the player can have an advantage. In this project, a simulation for the blackjack, and players for the blackjack are implemented. The machine learning agent for playing the game is implemented, tested, and results are compared. The results indicated that machine learning player can successfully replace the players utilizing the counting system.

ÖZET

MAKİNE ÖĞRENMESİ AJANI İLE YİRMİBİR OYUNU OYNANMASI

Yirmi bir oyunu, oyuncuya, dağıtıcıya karşı avantaj sağlama şansı veren bir kart oyunudur. Oyunun amacı, oyuncu elinin toplam değerinin 21 değerine eşit veya daha küçük olmasını sağlarken, dağıtıcının sahip olduğu toplam kart değerinden fazla olmasını sağlamaktır. Oyuncu, önceki ellerde oynanan kartların dağılımını da dikkate aldığında, değıtıcıya karşı bir avantaja sahip olabilir. Bu projede, yirmi bir oyunu için bir simülasyon ve simulasyon için oyuncular geliştirilmiştir. Yapay zeka ajanı oyunu oynamak için gelıiştirilmiş, test edilmiş ve sonuçlar karşılaştırılmıştır. Sonuçlar, makine öğrenmesi kullanan oyuncunun, sayma sistemi kullanan oyuncuların yerine başarıyla geçebileceğini göstermiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS / ABBREVIATIONS

BCF	betting criteria function
C_{Run}	running count
C_{True}	true count
CSV	comma-separated values
GUI	graphical user interface
r	the current run (e.g. numbered index of the run)
w	weights vector
v	observed count vector
«D»	set of cards to be dealt from the shoe during the game
λ	betting level
ϵ	ϵ-greedy action selection value

1. INTRODUCTION

Blackjack is used as a base for several different machine learning techniques in the past. Most of these techniques concerned about learning the basic strategy and some of them are used for learning to count cards. The aim of this project is to create a machine learning agent (machine learning player) to learn card counting and having an advantage over the dealer.

In this project, a blackjack simulation system is developed for the reason of testing different player strategies. This simulation receives parameters from the player; runs the test for given parameters; repeats the tests to find mean values and confidence intervals. The simulation is also used for creating a dataset to train the machine learning player. The dataset is created by monitoring the play of one of the card counting players.

The basic strategy is implemented as opposed to learned. This is done in order to create an even playfield for all players. The basic strategy is the optimal strategy for blackjack. Utilizing card counting methods can grant the player an advantage over the dealer.

The simulator is compared with a simulator from another paper to ensure that it’s working as expected. The statistics for basic strategy player and card counter players are presented. The machine learning player is trained using the dataset created by the simulator and tested under the simulation. It’s then compared with basic strategy player and card counter players.

2. BACKGROUND

2.1. Blackjack

The blackjack is a famous game played in different parts of the world. The standard 52-card deck is used for playing the game (de Mesentier Silva, Isaksen and Togelius 2). A game may have more than one deck of card and playing with 4, 6 or 8 decks is common. The game is played against the dealer. The players try to have a hand with higher total than the dealer without having a value a higher than 21. Except for the Jack, Queen, King, and Ace cards; the cards are at their face value. Jack, Queen, and King have a value of 10 and the ace has a value of either 11 or 1. Aces are worth 11 unless the total value exceeds 21. If the total value is more than 21, then the Aces are worth 1 and the hand is total is called a *soft* total.

At the start of the game, players place the bets and receive two face-up cards. The dealer receives two cards while one of them is face up and the other is facedown. The game continues as players make their move. The players have four different choices at that moment. The player can *hit* the card resulting in one more card to be dealt to the player. The player can *stand*, which the player stops making any moves until the end of the hand. The player can *double down* on the card, which doubles its bet; receives one more card and stops playing until the end of the hand. If the player has two cards with the same value, the player can *split* those hands. Splitting results in two new hands for the player to play. If the player exceeds the total value of 21, then the player has *bust*ed. Players can make plays until they stand or bust.

After every player make their play, the dealer plays. The dealer has a fixed rule which makes the dealer hit until the dealer's hand total is less than 17. When the soft total of 17 is reached, the dealer stops the play. The dealer can also bust by going over 21. Having a hand with an ace and a 10 valued card then it's called blackjack. A blackjack hand beats all hands except another blackjack hand. If the player has a blackjack and the dealer has either busted or stood below the count of 21, then the player wins 1.5 times its

bet. If the dealer hasn't busted and has a higher total than the player, then the player has lost. If the player has a higher total and the dealer has busted or has a lower total, then the player wins. Normally the dealer has an advantage over the player but if the player uses the basic Strategy and a card counting method, the player can have an advantage.

2.2. Related Works

The game of blackjack is an appealing game to test machine learning agents due to it chance factor. In this section, different machine learning techniques used on the game blackjack are discussed.

2.2.1. Blackjack as a Test Bed for Learning Strategies in Neural Networks

In this paper, blackjack is examined as a test bed for learning strategies in artificial neural networks (ANNs) with SARSA algorithm and reinforcement learning techniques (Pérez-Uribe and Sanchez 2022). The performance of fixed strategies, learned strategies and probabilistic strategies are tested in this paper. For simplifying the experiments; special features of the blackjack are removed ("insurance", "doubling down", "splitting pairs", etc.).

Reinforcement learning is a *system which learns by interacting with the environment*. During the learning phase; the system attempts some *actions* on the *environment*. Then the system is *reinforced* by getting a reward based on the consequences of those actions. The reinforcement algorithm works by repeating this in time steps. At every time step, the algorithm takes the state of the environment (s); chooses an action (a) to perform on the environment; on the next time step, it receives the reward (r); and it goes into a new state (s'). The system which implemented the reinforcement learning algorithm will try to maximize the reward it receives, by changing the actions in the next time step. SARSA (state-action-reward-state-action) algorithm is a reinforcement learning algorithm that takes its name from the quintuple (s, a, r, s', a').

Three fixed strategies are selected for testing: *dealer's strategy*, *hold strategy*, and a *random strategy*. Dealer's strategy is to *hit* until a card value total exceeds 17. Hold

strategy doesn't *hit* but *stands* (holds) at any value. Random strategy chooses its action randomly. Performance result of these strategies against the dealer's strategy is on average: dealer's strategy – 40.7%; hold strategy – 38.3%; random – 31.5% overall win percentages. These results suggest that matching the dealer's strategy on a blackjack game is the best fixed strategy. Even though the dealer's advantage is high and nearly 59.3% of games are lost.

Three different learning levels are used for the SARSA learning algorithm: $\epsilon = .1$, $\epsilon = .01$, and $\epsilon = .1 * 0.99^r$. This ϵ-greedy action selection value controls the *trade-off between exploitation and exploration ($\epsilon = .1$ has the highest exploration).* The *r* symbol indicates the current run. This algorithm is run 1000 times, so the number would start at 0 and increase until 1000 in each time step. Performance results of the learned strategy against the dealer's strategy are on average: $\epsilon = .1$ – 39.9%, $\epsilon = .01$ – 41.9%, and $\epsilon = .1 * 0.99^r$ – 40.9% overall win percentages. High exploration strategy finds excellent temporary solutions, but this may lead to decreased performance (lower than the dealer's fixed strategy). With probabilistic strategies, strategy changes in each time step. Performance result of the probabilistic strategy against the dealer's strategy is 49.14% on average, which is the highest result.

In conclusion; implementing the simple rules of a simplified version of blackjack was easy and the implemented code was not computationally intensive. Lastly, comparisons of performance are represented in the paper. It's important to note that, dealer's face-up card, which could give more information to the machine learning algorithm, were not considered.

2.2.2. The Evolution of Blackjack Strategies

In this paper, the evolution of a blackjack player is studied (Kendall and Smith 2474). Three neural networks are used to evolve the strategies of the blackjack player. A strategy is utilized where the worst networks are killed. At the end of this paper, the evolved strategy is compared with other famous blackjack strategies to show that it can best a typical blackjack player. It's also important to note that, the evolved player was able to learn parts of Thorpe's Basic Strategy.

Thorpe's work as cited by Kendall & Smith (2475) presented a system to give an average blackjack player an advantage over the casino. Thorpe's basic strategy is a table of rules which instructs what the player should do when a situation occurs. A classic example is when the player has a hard 16 and the dealer has is showing a 7 should the player draw another card or stand? The basic strategy can be easily learned and gives the player an edge over the dealer.

According to the authors, three distinct situations occur at a game of blackjack (splitting, doubling down and drawing a card), so three different neural networks are utilized instead of one. Splitting and doubling down networks requires three inputs (dealer's up card, player's two cards) with one hidden layer and one output. This network works only if the cards could be split/doubled down. Hit/stand network requires 16 inputs (dealer's up card and the maximum number of cards a player could have: 2, 2, 2, 2, 2, 2, A, A, A, A, A, A, A, A), 5 hidden nodes and one output. The networks can be seen in **Figure 2.1**.

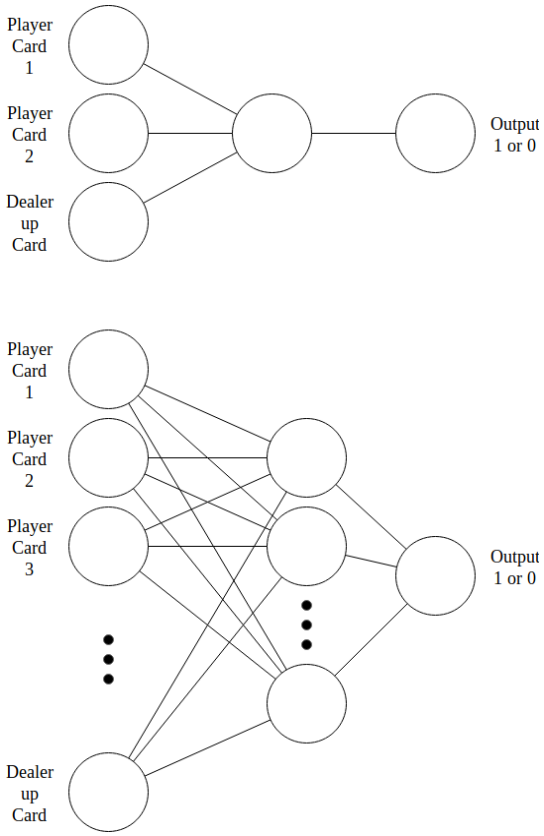

Figure 2.1 Architectures of doubling and splitting (top) and hit/stand networks (bottom)

An initial population consists of 60 players across 10 tables. Each table is dealt 1000 hands for each generation. The top 30 performers are transferred into the next generation while other 30 are killed. Remaining 30 are copied once to make the new population 60.

In blackjack, after doubling down, the player must take one more card and stand. For this reason, doubling down experiments are conducted in isolation. For this experiment, card values that couldn't be doubled down are ignored. The results of this experiment show that there is no learning for the doubling down network because the money amount stayed the same in each generation. Because the network didn't learn when to double down; the network is changed with a simple perceptron which yielded better results. After the change, AI players started to learn after 50 generations.

Normal play experiments are done by ignoring the splitting and doubling down actions. Players start to learn when to hit/stand after 100 generations.

For the reason that there is no immediate gain from splitting the cards; splitting experiments are done by running all three networks at the same time. Similar to the doubling down experiment, players didn't learn to split cards. So, architecture is simplified to a simple perceptron.

To compare the results, an average after 1000 hands are taken for the evolved player, Thorpe's strategy, match the dealer, and never bust strategies. Results are shown in **Figure 2.2**:

Figure 2.2 Comparison of blackjack strategies for evolving blackjack agents

The evolved player performs better than an average casino player and same as "match the dealer" and "never bust" strategies. However, evolved players still perform 5% worse than "Thorpe's strategy".

2.2.3. Boosting Blackjack Returns with Machine Learned Betting Criteria

In this paper, a technique to boost betting returns in Blackjack is investigated (Coleman 669). A genetic algorithm is used on betting criteria for outperforming standard criteria on ten different professional card counting systems. This paper also provides a metric for evaluating card counting algorithms.

In card counting systems, counts for the cards that are dealt from the shoe are generally assigned weights. Running count ($\mathbf{C_{Run}}$) is the dot product of the weights vector (**w**), and the observed count vector (**v**). The letter **i** is the index for the cards {i: 0 → Ace, 1 → "2"… 12 → "K"}. Which means, every card has a value (could be positive or negative) and when a card is dealt; its value is added to the C_{Run} where C_{Run} starts from zero whenever the shoe is reshuffled.

λ is the betting level, which is the largest absolute integer weight in a card counting system. If the largest absolute integer weight in the system is 0; then C_{Run} would also be 0, which in term means there is no card counting.

When the running count is adjusted to the shoe size and is normalized for comparison purposes; it's called the true count (C_{True}). To calculate it: $C_{True} = C_{Run}/(|«D»|\lambda)$ where the set of cards to be dealt from the shoe during the next hand is denoted as «D». To predict the player's advantage with max[•], the BCF (betting criteria function) is used. BCF is defined as follows: $BCF(C_{True}, \alpha, \beta) = \max[\alpha \times C_{True} - \beta, 0]$ where α and β are the betting criteria. For the standard criteria, α and β are 0.515% (0.00515) and 0.540% (0.00540) respectively. For the placed bet amount before the game begins K_j depends on both the Kelly's principle and the K_{min} (minimum house bet). The bet amount is calculated with the following: $K_j = \max[BCF(C_{True}, \alpha, \beta) \times F_j, K_{min}]$ where F_j is the current bankroll before the game. Above calculation means that the bet amount is calculated by taking the maximum value of the multiplication of the player advantage with the player's bankroll

and the minimum house bet. At the end of every game, the player's new bankroll is calculated by adding the outcome of the game to the old bankroll total.

3. ANALYSIS AND DESIGN

Project requirements, hardware/software requirements, hardware/software environments, general system design, software design, and counting system design, and machine learning agent design are explained in this section.

3.1. Project Requirements

Project requirements of the simulation system are as follows:

1. The programmed players must not exceed human capabilities while playing blackjack.
2. The simulation system must allow machine learning-players to play the game as well as programmed-players.
3. The simulation system must allow users to see the results of the simulations.
4. The simulation system must write the necessary information for training the machine learning-players, to a file.
5. The simulation system may write game information to a file for the user to view.
6. The system may allow users to train the machine learning player.

3.2. Hardware Requirements and Environment

This project requires a PC that can run the OS. The hardware environment of the development PC is: 3.2 GHz Intel Core 6500 Processor, 16GB of DDR4 RAM. Although moderately powerful hardware is used for development, much less powerful hardware could be used for this project. Using less powerful hardware will result in extended run-time for the simulation times; training and inference times for the machine learning player.

3.3. Software System Requirements and Environment

Software system requirements for this project are as follows:

1. The system must be able to simulate blackjack games.
2. The system should provide a GUI to the user for interaction.
3. The system must be able to take parameters for multiple experiments from the user.
4. The system must be able to present the simulation results to the user.
5. The system should present the simulation results to the user.

The software environment used for this project are:

1. Ubuntu GNU/Linux (operating system)
2. Python 3.6.7 (programming language)
3. Python3 pip (Python package manager to install modules)
4. NumPy (array-processing library), TkInter (GUI toolkit), pandas (data analysis library), SciPy (scientific and technical computing library), scikit-learn (machine learning library), matplotlib (plotting library), Jupyter
5. TensorFlow (machine learning framework)
6. Keras (high-level neural network API)
7. Oracle VM Virtual Box (virtual machine) (optional)
8. PyCharm (Python IDE) (optional)

3.4. Blackjack Simulator Software System Design

In this project, the simulation system should work on the personal computer of the user. The system must receive parameters for the simulations from the user, such as the number of decks to be used, the number of number of sessions to be simulated, number of hands per session, number of times the tests are repeated, starting bankroll, min & max bet amounts. The system then must simulate blackjack games for the given parameters. The use case diagram is shown in **Figure 3.1** for this system. During the simulation, the system should log the data to be used for training the AI agent, and necessary information for visualizing the simulation results. After the simulation, the system may present a table of results to the user.

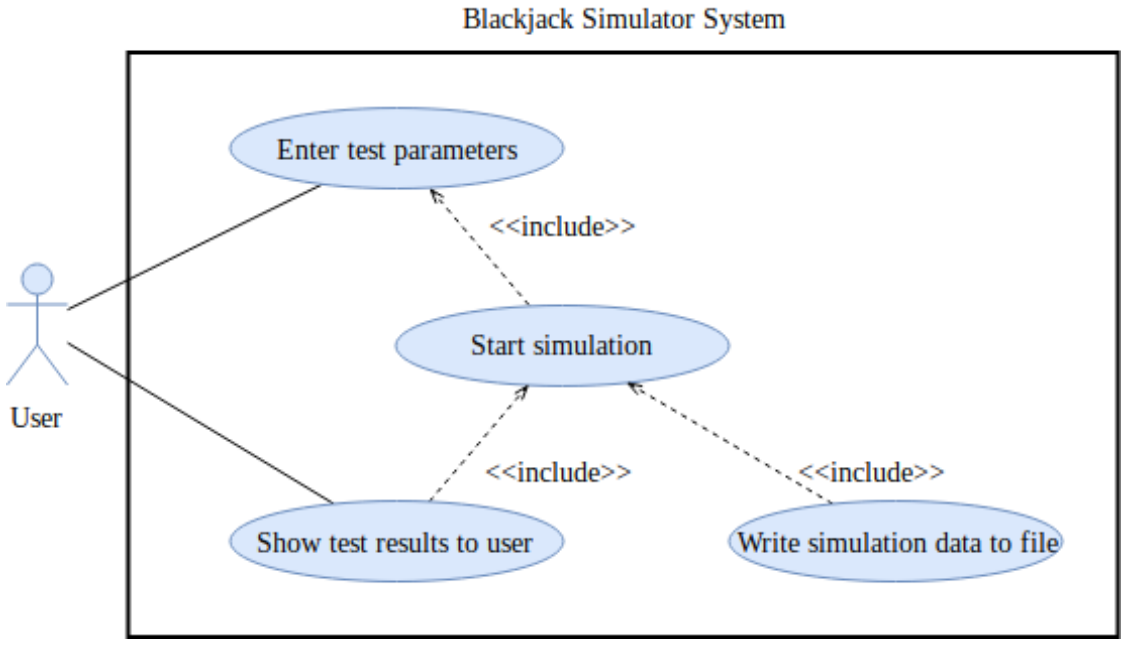

Figure 3.1 Blackjack simulation system's use case

The simulation is started by clicking the *Start Tests* button. Then the *Start Screen* is presented to the user. The user selects parameters such as the number of decks, the number of players, the number of hands per session, the number of sessions to be tested, the number of the tests are to be repeated, starting bankroll, minimum and maximum bet amounts. After the user enter the parameters, the tests are started by clicking the *Confirm*

Tests button. The *Results Screen* is presented to the user and the tests are created and run by that screen. Each time a test completed the results screen updated its contents to show the progress to the user. After all tests are complete, the user can press the Exit button to close the simulation. The event scenario can be seen on the **Figure 3.2**.

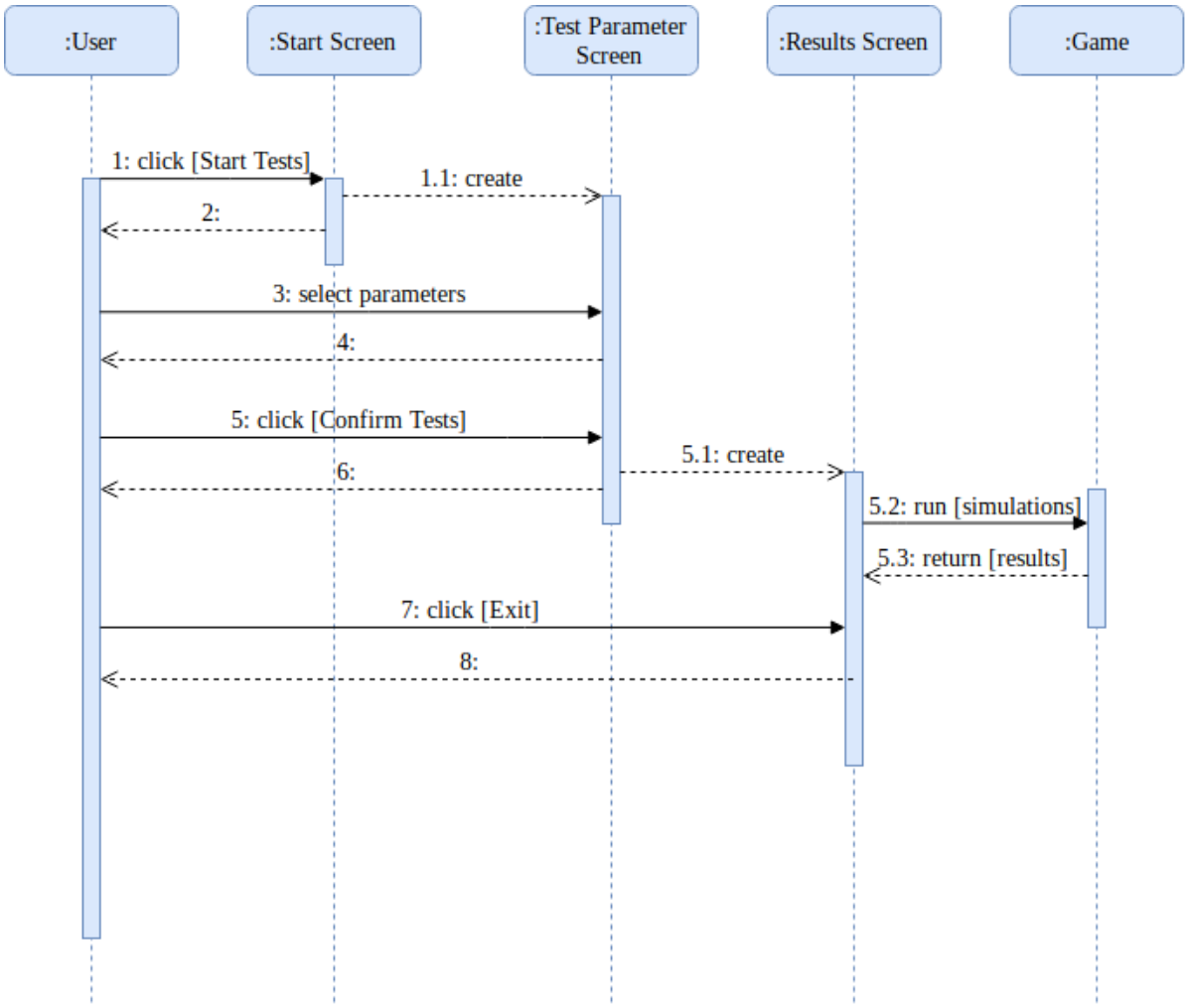

Figure 3.2 Event scenario for running tests in the blackjack simulation system

Blackjack games must be simulated according to the real-life blackjack rules. The simulation must implement a dealer, and the desired number of players to play the game. The cards must be dealt according to rules of blackjack and the blackjack games must be played in an order of a real-life blackjack game. An algorithm describing the outline of the blackjack game can be found in **Algorithm 1**.

Algorithm 1 An iteration for a game of blackjack in the simulation system

```
create a deck with 52 cards is created with integers 2 to 11
create a shoe by duplicating the deck and shuffle the shoe
set needs_reshuffle as false
while needs_reshuffle is false
  for every player
    open a hand for the player
    deal one card to the player
    advertise the dealt card to player
  end for
  deal one card to the dealer
  advertise the dealt card to card counters of every player
  for every player
    deal one card to the player
    advertise the dealt card to player
  end for
  deal one closed card to the dealer but don't advertise
  for every player
    for every hand of the player
      while players hand hasn't stood or busted
        make a choice between standing, splitting, or doubling down
      end while
    end for
  end for
  advertise the dealt card to card counters of every player
  while the dealer hasn't stood or busted
  if hand total of the dealer is smaller than 17
    deal one card to the dealer
    advertise the dealt card to card counters of every player
  end if
  end while
  for every player
    for every hand of the player
      if hand has busted
        the hand has lost
      else if hand has stood and the dealer is busted
        hand has won
        set winnings as bet
      else if hand and dealer have stood and hand total is more
        hand has won
        set winnings as bet
      else hand has lost
      end if
      if hand has won and hand is blackjack
        set winnings as winnings * 1.5
      end if
    end for
  end for
  if more than 80% of the deck is dealt
    set needs_reshuffle as true
  end if
end while
```

The blackjack games must be played, and tests must be repeated according to the amount of times specified by the user. After the tests complete, the advantage should be calculated according to a paper by Fogel (1428) which states, for an advantage of +0.13%, every bet of $100 by the player would mean a winning of 13 cents (0.13$) on average. So, the calculation of the advantage should be done by dividing the total of changes in the bankroll of the player with the total of the money the player has staked; for every repetition of a test.

This simulator should implement a basic strategy for simulated and machine learning player so that every player plays the same way. The advantage of using the basic strategy for all players is that, regardless of the cards played in the previous hands, all players make the same move. The basic strategy may be implemented as a lookup table, where along one edge of the table there are players' hands, and along another edge, there is dealer's card. All the players can look up this table to play a break-even strategy. The basic strategy for this project may be obtained through the Wizard of Odds website (Shackleford).

3.5. Counting Systems Design

Simulated players should use real-life card counting systems used by professional blackjack players. Card counting is used with the basic strategy to increase the players' advantage over the dealer. To achieve this; every player in the game should have a card counter which counts the values of advertised cards after the deck is shuffled. When a card is advertised, the card counter should check its corresponding value from a lookup table and add that value to the running count.

The true count should be calculated by estimating how many decks have remained in the shoe. This after this calculation a rounding operation may be done to simulate the way humans estimate the decks. Players must adjust their bets by the true count to increase their advantage. Counting systems may be implemented as an array indicating a plus or minus count to a corresponding card value.

3.6. Machine Learning Agent Design

During the simulation when a card is advertised to a card counter for the machine learning player (machine learning agent); the card counter may keep the cards in the memory. The machine learning player may then be able to use the data to infer the true count in the game.

The machine learning player may learn to count cards by mimicking a simulated player which uses real-life card counting methods. To achieve this, the simulated player's count information may be written to a file along with the distribution of cards in the game. If this method is used for the training, the count of each card should be recorded and normalized by the size of the shoe size (total number of cards in the game). Normalization should be done by dividing the count of each card with the shoe size. The true count value of the mimicked player and the normalized card counts should be written to a file for training. Inferring the true count should be done by giving normalized card counts (ten normalized card counts for each card value) as an input to the machine learning player and receiving the true count value. This true count should be used similarly as it's used for simulated players.

4. IMPLEMENTATION

The blackjack simulator software, counting systems, and machine learning agent implementations are explained in this section. This project is implemented on an Ubuntu Virtual Machine. PyCharm is used as the development environment, and Jupyter is used while developing machine learning agent. Python 3.6.7 is used as the programming language. Python is used as the programming language because of its support for Keras and other machine learning libraries. The machine learning model can be easily integrated into the simulation because both are written in the same programming language.

4.1. Blackjack Simulator Software System Implementation

The entry point for the simulator is in the *BlackjackSimulatorMain* module and this module contains GUI elements. The GUI for the blackjack simulator is implemented with the *TkInter* module for Python 3. The simulator gets the parameters for the simulation from the user. Before the starting the simulations, the user selects the number of decks, number of players; and enter the number of hands per session, number of sessions per test, number of times the tests are to be repeated, starting bankroll for the players, minimum and maximum bet amounts. A screenshot from the parameter selection screen of the simulator can be seen in **Figure 4.1**. The fields in the GUI is already filled for the user by default, so the user can see how the parameters are needed to be entered and it also provides a guideline for the user.

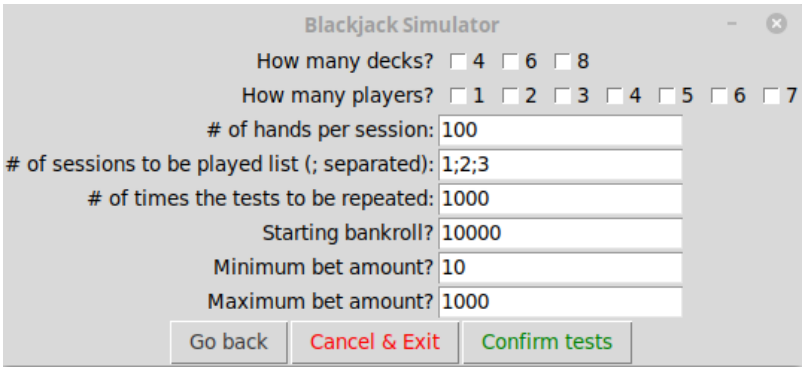

Figure 4.1 GUI parameter selection screen of the blackjack simulator

After changing the parameters, the user presses the "Confirm Tests" button to start the tests. Buttons for selecting the number of decks and players is implemented with *tkinter.Checkbutton*; text boxes for entering the other information is implemented with *tkinter.Entry*; labels are implemented with *tkinter.Label*. All this are placed in a grid inside of a *tkinter.Frame*. The buttons are implemented with *tkinter.Button* and placed in a frame. When the user confirms the tests by clicking the "Confirm tests" button, and error checking mechanism checks through the inputs the user has entered. If there exists an error in the input, the tests are not started, and the user is informed. While tests are running user sees a black screen indicating the tests are running.

When the tests complete, the user sees the results screen (**Figure 4.2**). The results screen contains simulation parameters on the leftmost side of the screen. For example, "1 session; 6 decks" indicates that the simulation used a 6-deck shoe and shuffled the shoe for every game in that session during the simulation. On the top, "Player x's (Card Counter Name) Δ%" denotes the percentage change in the bankroll of the player number x (indicated with a number from 0 through 6) with the name of the utilized card counter. The table is implemented with list of lists (2D list) of *tkinter.StringVar* variables in a grid of *tkinter.Label*'s. Every second column in the results screen denotes the player's calculated advantage for the tests by dividing the total change of bankroll with the total of bets placed by that player and taking the mean of advantage value for every repetition of tests.

The player advantage is calculated by the following formula:

$$advantage = \frac{\Delta bankroll}{\sum bets} \tag{4.1}$$

Where;

$\Delta bankroll =$ change in bankroll in one test simulation.
$bets =$ all the bets placed by the player in one test simulation.

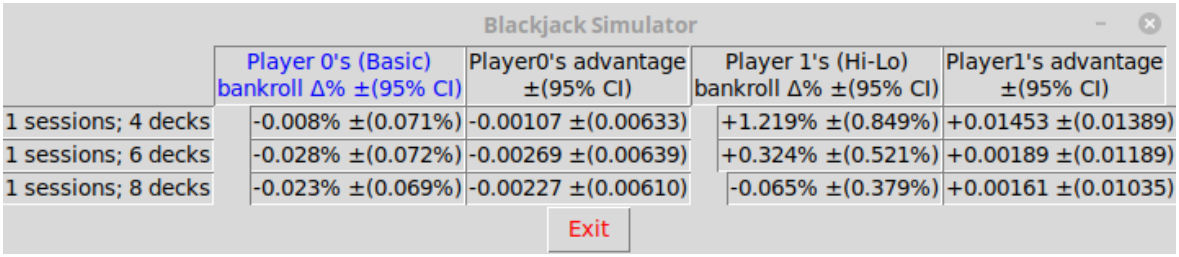

Figure 4.2 Results screen of the simulator for 4, 6, and 8 decks with 2 players

The 95% confidence interval is calculated and indicated in parenthesis next to the mean value of that player's advantage. The confidence interval is calculated by multiplying the z value for the 95% interval (1.960) with the division of standard deviation and square root of the number of times the experiment is repeated.

The simulator information for every game is recorded in a list of GameDataDump objects. This object record game information after the end of hand during the simulation for the following data:

- values of cards in the shoe
- a boolean value indicating whether the shoe needs to be shuffled
- minimum and maximum bet rules for the simulation
- number of decks used for playing the game
- the total number of players playing the game
- starting bankroll amount for players
- number of games and hands played up to that moment
- a list of counting systems used for by the players
- a list of players' calculated count information by their systems
- a list of players' remaining bankroll
- a list of players' total amount of bet placed from the beginning of the test
- a list for the distribution of played cards after the last shuffle and its corresponding true count value from a player for training the machine learning player

The information listed above is written to a CSV (comma separated values) file named *"count_list_for_ml.csv"*. Exporting the CSV file gives more ability to explore the simulations than the results screen.

When the tests start, a *Game* class is instantiated by the GUI's *run_one_test()* method for every different test configuration from the entered parameters. A *seed* parameter is given to the game object via parameters to seed the random number generator in the test. When the *Game* is run by calling its *start_game()* method, it instantiates a *Dealer*; the intended number of *Player* objects from the *Player* class; and an empty *numpy* array for holding the shoe. The *Game* is run multiple times to get the average test result. A generalized version of the class diagram for the blackjack simulator can be seen in **Figure 4.3**.

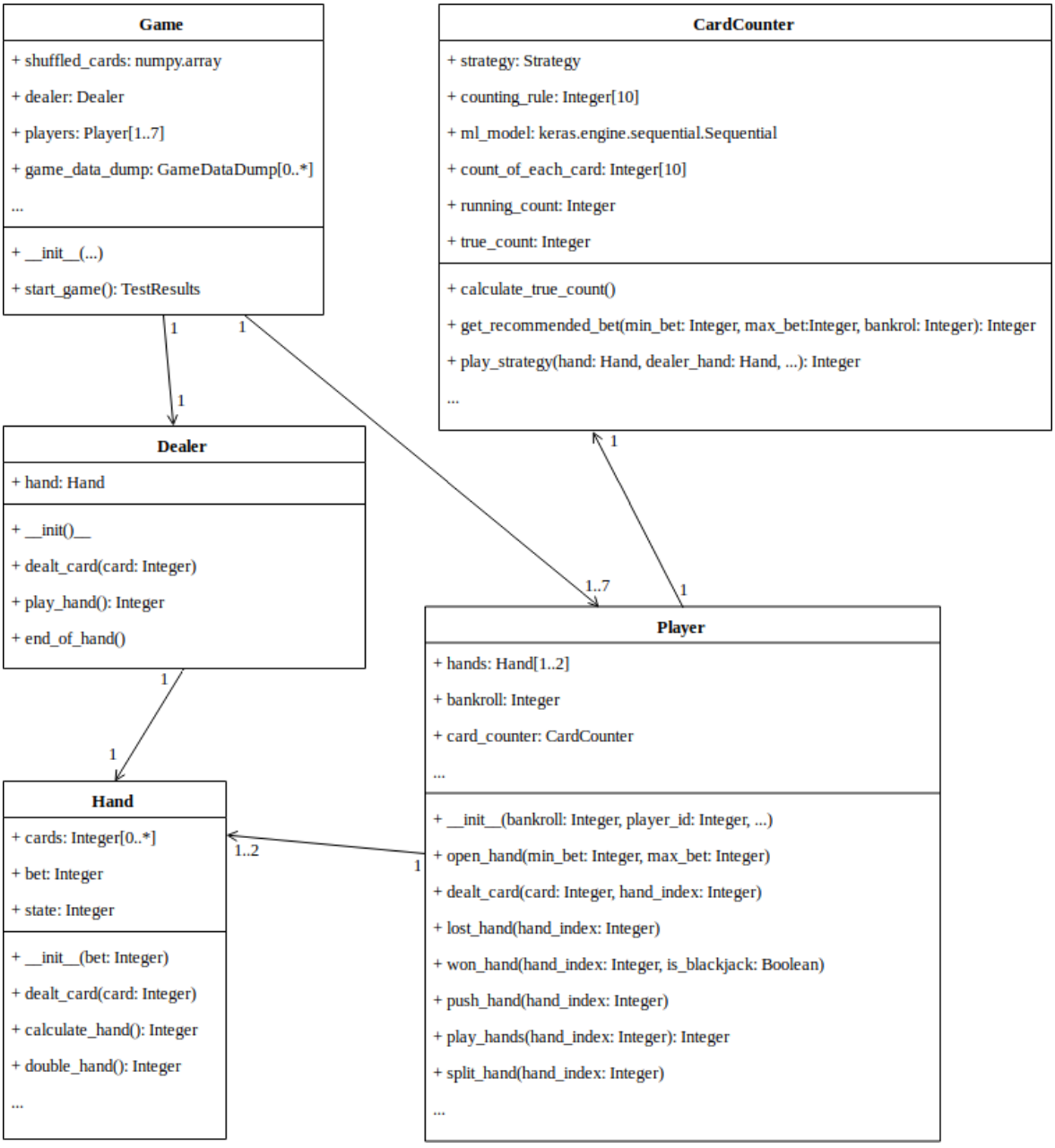

Figure 4.3 A simplified diagram showing classes of the blackjack simulator

The game creates a new shoe with the desired number of decks. It's accomplished by first creating a deck with four cards each with values 2 to 9, sixteen 10 value cards (10, J, Q, and K), and four 11 (Ace) values cards. A shoe containing multiple decks is achieved by copying a deck to the shoe with *numpy.append()*.

There are two main loops in the simulation: outer loop controls playing blackjack games, and inside that loop is the loop for playing blackjack hands. The loop is broken when the desired number of hands played by the simulation.

The outer loop starts by shuffling the deck with *numpy.random.shuffle()* method. It then resets *played_card_count* variable to zero and *needs_rehuffle* variable to *False*. The inner loop deals and plays blackjack hands until the cut card of the shoe is reached. The cut card is implemented as a floating-point number from 0 to 1 where values closer to 0 results in a smaller number of hands are dealt between shuffles and vice versa.

Inside the inner loop which runs while *needs_rehuffle* variable is false:

- Starting from player #0, all players are dealt one card from the shoe via calling players' *dealt_card()* method with the next card in the shoe.
- The dealt card is advertised to the card counters of every player via calling the card counters' *count_card_values()* method.
- The dealer is dealt one card via its *dealt_card()* method. The dealt card is again advertised to card counter of every player in the game.
- Each player gets one more card. The cards are advertised to card counters.
- The dealer is dealt one more card, but dealer's card is not added to the dealer's hand and the card is not advertised since the card needs to remain closed until later part of the hand. The card is stored at a variable called *dealer_second_card()*.
- For every player, the game is played for that player's hand(s) until the hand(s) either bust(s) or stand(s). Player's hand can hit (take more cards), double down (double its bet and draw one more card), split its hand (splits the hand to play each hand separately). Every card that the player has drawn from the shoe is advertised to card counters of every player. When the player completes playing the hand, next player starts to play until hand(s) of every player hand stood or busted.

- After the players play, the unadvertised card is dealt to the dealer and it's advertised to players' card counters.

- The dealer starts to draw cards until its hand reaches a value of soft 17 or busts (the dealer stands at soft 17).

- Hand(s) of every player is checked against the dealer's hand. If the players' hand exceeds a value of 21, then that hand has lost the bet. If the hand has stood and its value is the same as the dealer's hand value, then it's a draw. If it's greater than dealer's hand's value, then that hand has won the bet. If the player has won with a *Blackjack*, then the player wins 1.5 times the bet.

- The game information is calculated and recorded to the *game_data_dump* object before the game is reset for the next iteration of the inner loop.

- If the percentage of played cards is greater than the place of the cut card, then *needs_reshuffle* variable is set to *True*.

After the outer loop finishes its iterations; it gives the command to players to zero their card counters before the next reshuffle of the shoe. The outer loop continues until the number of blackjack hand played reaches the desired number given by the user at the parameter selection screen.

4.2. Counting Systems Implementation

Every player in the game has an instantiated *CardCounter* class. These counters all have an instance of *Strategy* class. The basic strategy is obtained from the Wizard of Odds website (Shackleford) and written into a CSV file called *basic-strategy.csv*. In the strategy, the *basic-strategy.csv* CSV file is read and loaded into a *numpy.matrix*. This list contains the basic strategy for the players. There is also a *play_strategy()* method which takes the player's hand and the dealer's hand as parameters and looks up the basic strategy to return an *enum* which denotes to a move a player can make. The actions returned by the method are 101 for HIT, 102 for SPLIT, 103 for DOUBLE, and 100 for STAND.

Card counters utilize card counting systems besides basic strategy. Card counters get a vector of weights from the *CountingSystems* module. This module provides a lookup for a counting system name and returns a numpy array consisting of 10 integers. These integers donate cards from 2 to 9, 10 values cards (K, J, Q), and ace card. A table of weight vectors of implemented Hi-Lo card counter can be seen from the **Table 4.1** (Coleman 672). Integers in the array denote to the value a card should add to the running count. Card counters get advertised of dealt cards whenever their *count_card_values()* method gets called. This method uses the array to find the value a card adds to the count and adds that value to the *running count*. The running count is calculated when a bet recommendation is requested by calling the *calculate_true_count()* method.

Table 4.1 Weight vectors for counting systems

	A	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**T**
No card counter	0	0	0	0	0	0	0	0	0	0
Hi-Lo	-1	1	1	1	1	1	0	0	0	-1

True count is calculated by estimating how many decks have remained in the shoe. This estimation is done by subtracting the number of played cards (*played_card_count* variable) from the number of total cards in the shoe (*total_cards_in_shoe* variable). Because humans can't keep track of every card in the deck; this estimation is rounded by a floor operation to simulate human behavior. This is calculated by dividing the remaining card count by the number 26 to find how many half decks are in the shoe (a deck has 52 cards and a half deck has 26 cards). Floor operation is applied to this value and subsequently divided by the number 2 to produce an estimated value for remaining decks. The remaining deck value is multiplied by the betting level of the card counter and the resulting value is used to divide the *running_count* to produce the true count for the game. The betting level is the maximum value of cards in the numpy array consisting of integers.

The following formula is used to calculate the true count:

$$remaining\ decks = \frac{floor\left(\frac{|\text{«D»}|}{26}\right)}{2} \tag{4.2}$$

$$C_{True} = \frac{C_{Run}}{(remaining\ decks)\lambda} \tag{4.3}$$

Where;

C_{True} = true count of the card counter.
C_{Run} = running count of the card counter.
«D» = remaining cards in the shoe.
λ = betting level of the card counting strategy.

To get the recommended bet, *get_recommended_bet()* method of the CardCounter class is called by the player. This method first calls its *calculate_true_count()* method to get a true count estimation for the distributed cards in the game. It's assumed while implementing the card counters, the advantage of a freshly shuffled deck has an advantage of -0.5% (-0.005) and the advantage increases by 0.5% (0.005) for each increase in the true count. The player advantage is calculated by multiplying true count by 0.005 and subtracting 0.005 from the result.

The following formula is used to calculate the recommended betting amount:

$$bet\ amount = trunc\big((0.005C_{True} - 0.005)bankroll\big) \tag{4.4}$$

Where;

C_{True} = true count of the card counter.

Bet amount is calculated by multiplying the previously found advantage with the player's bankroll amount and truncating the resulting value to an integer. If the resulting value is greater than the maximum possible bet by the game's rules, the maximum value is assigned to the variable. If the result is smaller than the minimum possible bet, then

minimum bet is returned to the caller instead of the calculated value of the recommended bet.

4.3. Machine Learning Agent Implementation

In the blackjack simulator, machine learning capability is implemented in *count_card_value()* and *calculate_true_count()* methods of the *CardCounter* class. In the *count_card_value()* method, a *numpy.array* called *count_of_each_card*, consisting of 10 integers for card values keeps track of distributed cards in the game. Cards with values 2 through 9 is kept at indexes 0 through 7, 10 valued cards are kept at index 8, and 11 valued ace cards are kept at index 9 of the array. Each time a card is advertised to this method, corresponding element of the array gets incremented by one. This array is used both for training the machine learning player and inferring the true count by that player.

When creating the CSV file for the training of the machine learning player a true count value taken from a simulated player is appended to the *CardCounter*'s *count_list* list in addition to the *count_of_each_card* array. Before the array is written, the values are normalized by dividing each element with the total number of cards in the game (*total_cards_in_shoe*). This operation is executed in the *get_recomented_bet()* method of a simulated player. The player is being recorded for the purpose of creating data for training. A player is chosen if its name matches the variable in the *Settings* module called *COUNT_SYSTEM_TO_WATCH*. In the *Game* object, this *count_list* list is returned to the calling function as a part of a *TestResults* object. When the list returns to the run_one_test() method, contents of the list is written to a file called "*count_list_for_ml.csv*".

The machine learning player's model is implemented to receive the normalized card counts as input and true count as output. The model is instantiated as a *Sequential* Keras model and added an LSTM layer with *20 neurons* and an input shape of *1, 10*. A dense layer with *1 neuron* with no activation function is also added to the model. Input and output shapes for the model can be seen in **Figure 4.4**.

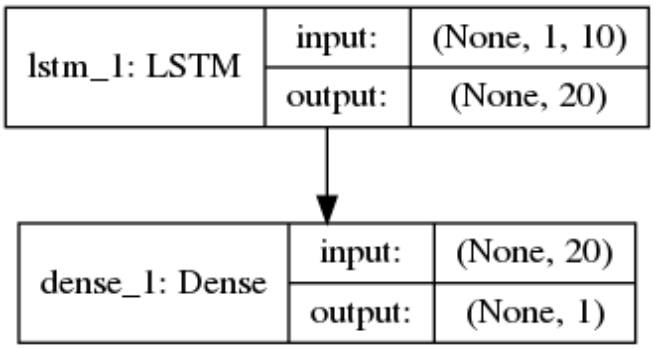

Figure 4.4 The input and output shapes of the neural network model

The model is then compiled with *mean squared error* loss function, and *adam* optimizer. All other parameters are left default. A visualization for the compiled model can be seen in **Figure 4.5**. The first layer contains 2480 trainable parameters and the second layer contains 21 trainable parameters which result in a total of 2501 trainable parameters.

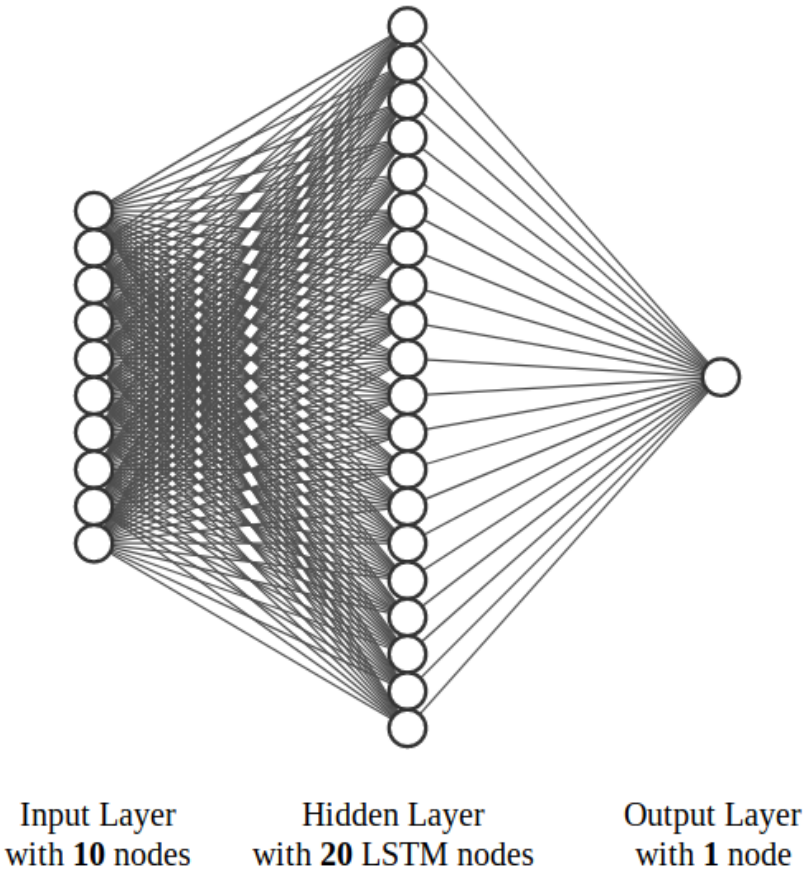

Figure 4.5 Visualization for the neural network model

For creating the dataset for training, the simulations are started with 2 players per game (basic strategy player and card counter with Hi-Lo strategy); 4, 6, and 8 decks; 100 hands per session; 1 session per test; 100; and tests are repeated 100 times. During the simulations, the Hi-Lo counting strategy its true count along with the normalized distribution of cards are recorded to the CSV file. The file is read by the *TrainML.py* and its contents are loaded into a *pandas dataframe*. Rows with the name *card2, card3… card11* are used as input variables and *f_count* variable is used as the output variable. The dataframe is shuffled and split into train and test sets with a test size of fraction *0.3* i.e. 30% of the set is set aside for testing.

The 30% of the training data is set aside for the validation and rest is used for training the model. Monitoring the validation loss parameter, the model is trained with the training set and stopped when validation loss stopped decreasing for 2 epochs. The loss is calculated as the mean squared error. The best model is saved during the training. The mean squared error on the train and test sets are 0.3, and the R^2 scores for training and test are 0.950 and 0.955 respectively. The accuracy and the validation accuracy scores for the best model is 0.7155 and 0.7380 respectively.

After the machine learning model is created and trained, it's loaded into the simulator with the *load_model()* method from *keras.models*. When the *calculate_true_count()* method is called for the machine learning player, the *count_of_each_card* array is normalized. It's then added another dimension to be compatible with the model. After preparing the array, the model predicts the true count of the game by using its *predict()* method on the array. The resulting prediction is truncated and cast into an integer.

The *count_of_each_card* array is reset by the player before shuffling the cards. This operation is done by calling *CardCounter.zero_counter()* method from the *Player*'s *end_of_game()* method.

5. TEST AND RESULTS

5.1. Test of Blackjack Simulator Software

The tests are conducted by having the same random set of seeds for every setting of the experiment. This ensured that every player plays the same game as other players have with the same cards during the tests to avoid variables caused by playing the different cards between players.

5.1.1. Purpose

In this test, blackjack simulator software is compared to another simulation system. The purpose of this is to verify that the simulated software is a realistic representation of other simulation systems.

5.1.2. Procedure

The basic strategy is tested with 4 decks; 100 hands per session, 1 session per play, starting bankroll of 10000, betting range between 10 and 1000. The tests are repeated 10000 times and average bankroll change percentage, player's advantage with 95% confidence interval is recorded. The additional rules for this simulation are: the cut card is placed at the 80% of the shoe, dealer stands on soft 17, double after splitting is allowed, no insurance allowed, hands can be split once. The results are compared with the different strategy results described in a paper by Fogel (1429). In the paper, *Revere 4-Decks* strategy has an advantage of -.00436 ±(.00115); *Archer 4-Decks* has an advantage of -.00433 ±(.00115); *Gollehon 4-Decks* has an advantage of -.00431 ±(.00115); and the *Patterson 4-Decks* has an advantage of -.00705 ±(.00115). It's expected that the simulation and the basic strategy player to behave have a similar advantage.

5.1.3. Findings and Comments

Basic strategy test results can be seen in **Table 5.1**. The basic strategy in the simulation has a similar advantage to the strategies in Fogel's paper (1429). The simulation

on that paper is a simulation. This suggests that the blackjack simulation in this project is behaving realistically. It can be seen that using just a basic strategy has a slight disadvantage.

Table 5.1 Test results for basic strategy with 4 decks

	Basic strategy mean player bankroll Δ% ±(95% CI)	Basic strategy mean player advantage ±(95% CI)
1 session with 4 decks	-.065% ±(.022%)	-.00607 ±(.00198)

5.2. Test of Counting Systems

5.2.1. Purpose

In this test card counting strategy of simulated players are compared with players only using basic strategy. The purpose of this experiment is to measure the performance difference between a player using basic strategy but no card counting to a player using basic strategy and card counting.

5.2.2. Procedure

In this test, first, the basic strategy player plays the blackjack games by just itself (no other players at the blackjack table) with the rules and parameters from the previous test for 4, 6, and 8 deck games. After basic strategy tests complete, card counting players are tested (only one player per test) with the same parameters and the results are recorded.

5.2.3. Findings and Comments

The test results between the card counter player and the basic strategy can be seen in **Table 5.2**. The Hi-Lo card counting strategy on average has a positive advantage and the basic strategy has a negative advantage for all test settings. The card counter player also has a much positive bankroll on the average. The confidence interval for the counting

player is wider because the player can increase its bet by the true count and can still lose the bet. It's also true that using a counting method in addition to a basic strategy has an advantage over just using the basic strategy.

Table 5.2 Test results for basic strategy and card counter players on 4, 6, and 8 decks

	Basic strategy mean player bankroll Δ% ±(95% CI)	Basic strategy mean player advantage ±(95% CI)	Hi-Lo strategy mean player bankroll Δ% ±(95% CI)	Hi-Lo strategy mean player advantage ±(95% CI)
1 session with 4 decks	-.065% ±(.022%)	-.00607 ±(.00198)	+.671% ±(.249%)	+.00530 ±(.00409)
1 session with 6 decks	-.066% ±(.022%)	-.00609 ±(.00198)	+.147% ±(.164%)	+.00087 ±(.00347)
1 session with 8 decks	-.043% ±(.022%)	-.00414 ±(.00196)	+.234% ±(.113%)	+.00392 ±(.00292)

5.3. Test of Machine Learning Agent

5.3.1. Purpose

In this test, the machine learning player is compared with simulated the player which uses basic strategy and compared with the player which uses the card counting strategy. The purpose of this experiment is to measure the performance difference between machine learning player and basic strategy player, and between machine learning player and card counter player.

5.3.2. Procedure

In this experiment, machine learning is tested with same parameters and rules are used in the previous tests, playing the blackjack games by just itself. The tests are again repeated 10000 times and average bankroll change percentage, and player's advantage with 95% confidence interval is recorded.

5.3.3. Findings and Comments

The test results of the machine learning player strategy can be seen in **Table 5.3**. The results for basic strategy and card counting methods can be seen in **Table 5.2**. The machine learning player's advantage and increase in bankroll is considerably higher than basic strategy player.

Table 5.3 Test results for machine learning player on 4, 6, and 8 decks

	Machine learning player mean bankroll Δ% ±(95% CI)	Machine learning player mean advantage ±(95% CI)
1 session with 4 decks	+.422% ±(.164%)	+.00561 ±(.00358)
1 session with 6 decks	+.112% ±(.125%)	+.00118 ±(.00318)
1 session with 8 decks	+.187% ±(.095%)	+.00330 ±(.00279)

The average advantage of the card counter and machine learning players are nearly identical, and both advantages are better than basic strategy. The machine learning player has a slightly better advantage than the card counter player in 4 and 6 decks. In 8 deck games, the card counter has a slightly better advantage. This difference may be due to the card counter player estimates the remaining decks while the machine learning player doesn't do any estimation.

There is a difference between the card counter and machine learning players in the change in the bankroll amount. On average the machine learning player has increased its bankroll amount less than the card counter player. This means that machine learning player has put a lower total bet amount at stake than the card counter player which impeded the potential gain of the bankroll of the machine learning player.

6. CONCLUSION

This project is interested in the problem of training a machine learning player, where it learns to find the true count value for the distribution of cards in the simulation. A blackjack simulator is developed in order to train and test the machine learning player. The blackjack simulation is tested against another lifelike simulation and the results are compared. The comparison between two simulations suggests that the simulation is realistic and the test results for testing the basic strategy, card counting player, and machine learning player are valid.

The basic strategy player almost evens the odds of the player with the dealer. So, every player is using the basic strategy as a base for card counter and machine learning player. Because every player has the advantage of using the basic strategy, altering the bets according to the distribution of cards method allows players to increase their advantage and allow easier comparison between different players.

The results show that the machine learning player is comparable to a real-life counting system Hi-Lo and it can learn to produce true counts to alter the bet and have an advantage over the dealer. The machine learning player can successfully replace the counting procedure.

Bibliography

Coleman, Ron. "Boosting Blackjack Returns with Machine Learned Betting Criteria." *Proceedings of the Third International Conference on Information Technology: New Generations (ITNG'06)*, Las Vegas NV, USA: IEEE, 2006. pp. 669–673. doi:10.1109/ITNG.2006.40.

de Mesentier Silva, Fernando, et al. "Generating Heuristics for Novice Players." *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. Santorini, Greece: IEEE, 2016. pp. 1–8. doi:10.1109/CIG.2016.7860407.

Fogel, David B. "Evolving Strategies in Blackjack." *Proceedings of the 2004 Congress on Evolutionary Computation CEC 2004*. Portland OR, USA: IEEE, 2004. pp. 1427–1434. doi:10.1109/CEC.2004.1331064.

Kendall, Graham and Craig Smith. "The Evolution of Blackjack Strategies." *The 2003 Congress on Evolutionary Computation: CEC 2003*. Canberra, Australia: IEEE, 2003. pp. 2474–2481. doi:10.1109/CEC.2003.1299399.

Pérez-Uribe, Andrés and Eduardo Sanchez. "Blackjack as a Test Bed for Learning Strategies in Neural Networks." *The 1998 IEEE International Joint Conference on Neural networks proceedings IEEE World Congress on Computational Intelligence*. Anchorage, Alasky: IEEE, 1998, pp. 2022–2027. doi:10.1109/IJCNN.1998.687170.

Shackleford, Michael. "4-Deck to 8-Deck Blackjack Strategy." *Wizards of Odds*, 27 June 2016. wizardofodds.com/games/blackjack/strategy/4-decks/. Accessed 30 December 2018.

Thorp, Edward O. *Beat the Dealer. A Winning Strategy for the Game of Twenty One*. New York: Vintage Books, 1966.