**Two Shuffling Techniques in Poker**

Geoffrey Wong 1155109054

Chao Yu 1155053722

1. Introduction

**1.1 Project Background**

In many strategy-based games, a randomizer is often needed in order to keep matches independent from previous ones. A good example is poker. If the games are run by a computer, they have access to a built-in RNG that usually does the job. However, in real-life scenario, we do not have such luxury and often need to resort to some tricks. In this project, we will look into two most prominent and frequently-used techniques, overhand shuffle and riffle shuffle, for introducing randomness into poker deck. The shuffling techniques are first modeled in Python programs, and their effectiveness will be evaluated by mathematical models. In addition, we also looked into some strategies for card-guessing given that the way of shuffling is known through simulations. The results of the simulations will be presented.

Before we dive into the specific domains of overhand shuffle and riffle, we will first discuss some general topics of card shuffling, which would prove useful later on.

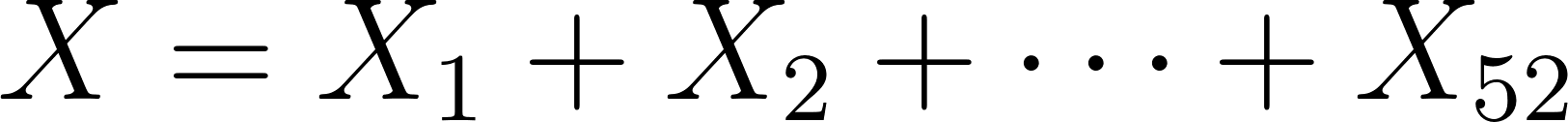
1.2 Experimental Evaluation of Randomness

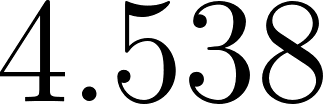
A simple way to evaluate randomness of a shuffling technique is by repeated experiments. The experiment involves 2 participants, a guesser and a shuffler. The game begins with a properly ordered deck of poker cards. For convenience, they are referred to as number 1 to 52. Then, the shuffler shuffles the deck in a predetermined way that is known by both participants (i.e. riffle shuffle 3 times). After that, the game will be played according to the following procedures:

1. Guesser makes a guess on what the top card is.
2. The top card is revealed and thrown to a discard pile.
3. The number of correct guesses are recorded.

These steps are repeated until no more cards are left in the deck.

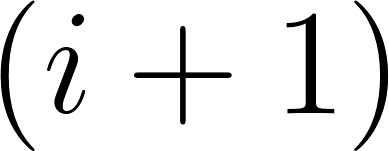
If the guesser guesses randomly, the expected value of correct guesses is 1.

If the guesser eliminates the possibility of the cards from the discard pile and guesses randomly from the remaining cards, we denote the number of correct guesses on a 52-card deck as a random variable [](https://www.codecogs.com/eqnedit.php?latex=X%20%3D%20X_1%20%2B%20X_2%20%2B%20%5Cdots%20%2B%20X_%7B52%7D%0) where [](https://www.codecogs.com/eqnedit.php?latex=X_i%0) takes [](https://www.codecogs.com/eqnedit.php?latex=1%0) if the [](https://www.codecogs.com/eqnedit.php?latex=i%0)-th guesses is correct and [](https://www.codecogs.com/eqnedit.php?latex=0%0) otherwise. By the linearity of expectation, we get [](https://www.codecogs.com/eqnedit.php?latex=E%5BX%5D%20%3D%20E%5BX_1%5D%20%2B%20E%5BX_2%5D%20%2B%20%5Cdots%20%2B%20E%5BX_%7B52%7D%5D%20%3D%201%2F52%20%2B%201%2F51%20%2B%20%5Cdots%20%2B%201%2F1%20%5Capprox%204.538%0)

If the guesser managed to derive information from the way the cards are shuffled, which means the shuffling technique is not completely random, the average correct guesses should be higher than [](https://www.codecogs.com/eqnedit.php?latex=4.538%0).

The flaw of this method, however, is that it does not guarantee that the guesser is playing the game with the best strategy and it takes too much resources.

1.3 Increasing Sequences

When shuffling an initially sorted deck, a good technique to evaluate its “randomness” is to track its increasing sequence. We define an increasing sequence as a subsequence of the deck, where the card number in the [](https://www.codecogs.com/eqnedit.php?latex=i%0)-th position is less than the card number in the [](https://www.codecogs.com/eqnedit.php?latex=(i%2B1)%0)-th position by exactly [](https://www.codecogs.com/eqnedit.php?latex=1%0) for all cards in the subsequence. For example, in a 10-card deck:

* [1,2,3,4,5,6,7,8,9,10] has only one increasing sequence: [1,2,3,4,5,6,7,8,9,10].
* [1,2,7,3,4,8,9,5,10,6] has two increasing sequences:[1,2,3,4,5,6], [7,8,9,10].
* [4,1,2,8,9,7,5,10,3,6] has four increasing sequences:[1,2,3],[4,5,6],[7],[8,9,10].

Note that a subsequence with a single element is also counted as an increasing sequence.

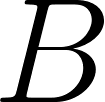
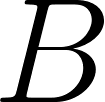
2. Overhand Shuffle

2.1 Introduction

Overhand shuffling is one of the most easiest shuffling methods which the shuffler slides out a small packet of cards with random size from the middle portion of the deck, and place the packet back on top of the deck.

2.2 Our Model

We formulate Overhand Shuffle as thus:

1. Pick a card [](https://www.codecogs.com/eqnedit.php?latex=A%0) uniformly random from the deck, excluding the first and the last card.
2. Pick a card [](https://www.codecogs.com/eqnedit.php?latex=B%0) uniformly random from the next card of [](https://www.codecogs.com/eqnedit.php?latex=A%0) to the last card of the deck.
3. Take out the packet of cards from [](https://www.codecogs.com/eqnedit.php?latex=A%0) to [](https://www.codecogs.com/eqnedit.php?latex=B%0) and place them back on the top of the deck.

Below is the Python code to simulate the outcome of this model.

|  |
| --- |
| **if** self.style == "Overhand Shuffle":  a = random.randint(1, 51)  b = random.randint(a+1, 52)  self.cards = self.cards[a:b+1] + self.cards[:a] + self.cards[b+1:] |

2.3 Analysis

In the paper by Johan Jonasson, it is stated that the mixing time of overhand shuffle is . Therefore, for a deck of poker of 52 cards, it takes approxiately 52\*52\*log(52) = 4640 shuffles to obtain a uniformly random deck.

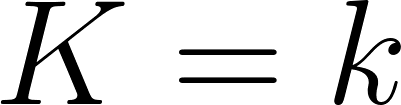
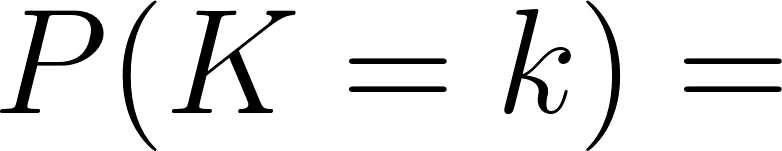
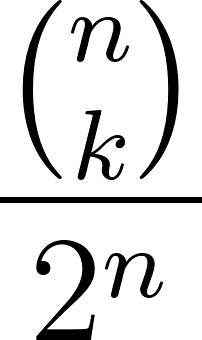
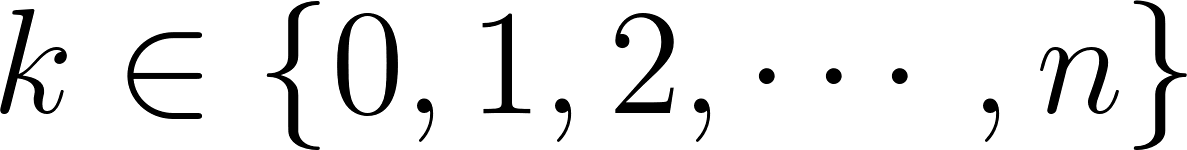
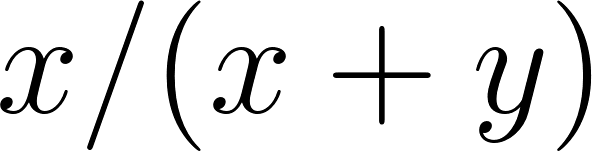
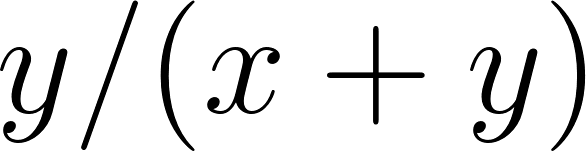
3. Riffle Shuffle

3.1 Introduction

Riffle Shuffle is a common technique which roughly half of the deck is held in each hand with thumb pressed inward. Then the cards are released by the thumb so they fall together back to an interleaved deck.

3.2 The Gilbert-Shanon-Reeds Model

Riffle Shuffle was modelled mathematically by Gilbert in 1955 with the name Gilbert-Shannon-Reeds (GSR) Model [1]. It has been verified as a good match for outcomes of human shuffling by empirical evidence. The model is formulated as thus (assuming the deck contains [](https://www.codecogs.com/eqnedit.php?latex=n%0) cards):

1. The deck is divided into two packets, with [](https://www.codecogs.com/eqnedit.php?latex=k%0) and [](https://www.codecogs.com/eqnedit.php?latex=n-k%0) cards respectively. [](https://www.codecogs.com/eqnedit.php?latex=K%20%3D%20k%0) is a random variable which [](https://www.codecogs.com/eqnedit.php?latex=P(K%20%3D%20k)%20%3D%0) [](https://www.codecogs.com/eqnedit.php?latex=%5Cfrac%7B%7Bn%20%5Cchoose%20k%7D%7D%7B2%5En%7D%0), where [](https://www.codecogs.com/eqnedit.php?latex=k%20%5Cin%20%5C%7B0%2C%201%2C%202%2C%20%5Ccdots%2C%20n%5C%7D%0).
2. The cards are moved from the bottom of the two packets one at a time, with [](https://www.codecogs.com/eqnedit.php?latex=x%2F(x%2By)%0) and [](https://www.codecogs.com/eqnedit.php?latex=y%2F(x%2By)%0) probability of choosing from the left packet and right packet respectively, when the left packet contains [](https://www.codecogs.com/eqnedit.php?latex=x%0) cards and the right packet contains [](https://www.codecogs.com/eqnedit.php?latex=y%0) cards.

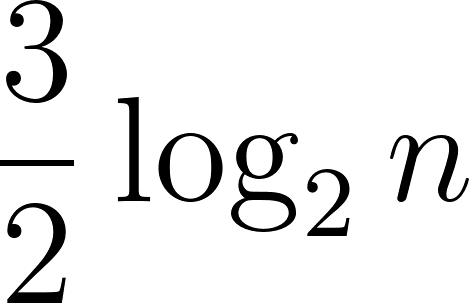
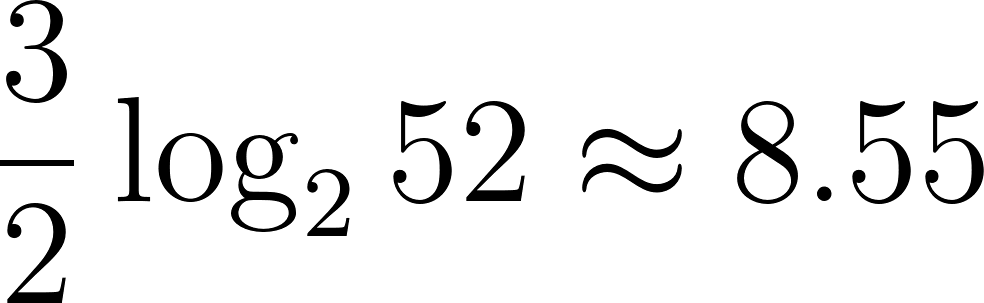
Aside from the definition above, there is a more convenient procedure to generate the identical outcomes with the exact same probabilities. We will simulate riffle shuffle using these set of steps:

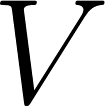
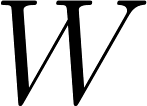
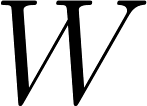
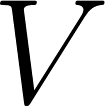
1. Flip a fair coin [](https://www.codecogs.com/eqnedit.php?latex=n%0) times to determine the size of the left packet and the right packet. The size of the left packet equals to the total number of heads and the size of right packet equals to the total number of tails (or vice versa).
2. Now use the same outcomes of the coin tosses in sequence to determine which packet to choose to fill in the position. That is, if the outcome appears to be heads, we take a card from the left packet and place it in the corresponding position; If the outcome appears to be tails, we take a card from the right packet.

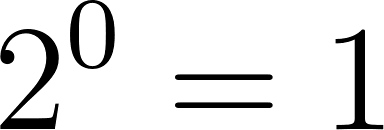
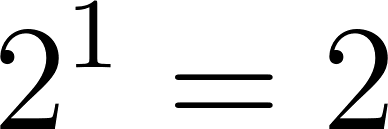
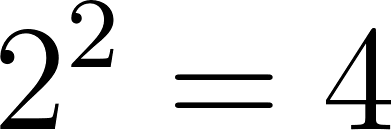
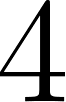
Below is a Python code to simulate the outcome of GSR model of a 52-card deck, using the previous steps.

|  |
| --- |
| **if** self.style == "Riffle Shuffle":  coin\_flips = [random.randint(0, 1) **for** \_ **in** range(len(self.cards))]    heads = coin\_flips.count(1)  tails = coin\_flips.count(0)    left\_packet = self.cards[:heads]  right\_packet = self.cards[heads:]    result = []  **for** coin **in** coin\_flips:  **if** coin == 1:  result.append(left\_packet[0])  left\_packet = left\_packet[1:]  **else**:  result.append(right\_packet[0])  right\_packet = right\_packet[1:]    self.cards = result |

3.3 Analysis

In a paper written by Bayer and Diaconis, it is stated that the expected shuffles to uniformly randomize a deck of [](https://www.codecogs.com/eqnedit.php?latex=n%0) cards is around [](https://www.codecogs.com/eqnedit.php?latex=%5Cfrac%7B3%7D%7B2%7D%5Clog_%7B2%7Dn%0) when applying riffle shuffle [2]. For a deck of 52 cards, [](https://www.codecogs.com/eqnedit.php?latex=%5Cfrac%7B3%7D%7B2%7D%5Clog_%7B2%7D52%20%5Capprox%208.55%0), meaning we have to perform riffle shuffle around 8 to 9 times to grant a randomized deck. The details of the result is mathematically involved and lengthy, but the basic idea is that we cannot reach certain permutations of a deck if we shuffle less than 8 to 9 times.

Observe for each riffle shuffle, we can transition from permutation [](https://www.codecogs.com/eqnedit.php?latex=V%0) to permutation [](https://www.codecogs.com/eqnedit.php?latex=W%0) where the number of increasing sequences of [](https://www.codecogs.com/eqnedit.php?latex=W%0) is at most two times the increasing sequence of [](https://www.codecogs.com/eqnedit.php?latex=V%0). For example, below is a possible process when riffle shuffling a deck of 10 cards for three times.

* Initial State: [1,2,3,4,5,6,7,8,9,10]. It has [](https://www.codecogs.com/eqnedit.php?latex=2%5E0%3D1%0) increasing sequences.
* First Shuffle: [1,2,7,3,4,8,9,5,10,6]. It has [](https://www.codecogs.com/eqnedit.php?latex=2%5E1%3D2%0) increasing sequences: [1,2,3,4,5,6], [7,8,9,10].
* Second Shuffle: [4,1,2,8,9,7,5,10,3,6]. It has [](https://www.codecogs.com/eqnedit.php?latex=2%5E2%3D4%0) increasing sequences:[1,2,3],[4,5,6],[7],[8,9,10].
* Third Shuffle: [4,7,1,2,5,10,8,3,6,9]. It has [](https://www.codecogs.com/eqnedit.php?latex=4%0) increasing sequences: [1,2,3],[4,5,6],[7,8,9],[10].

It is impossible to reach permutations with 4 increasing sequences after a single shuffle from the initial state, so it is obvious that a single shuffle of a deck of 10 cards is ineffective.

4. Machine Learning Agent

We begin our search for the optimal guessing strategy with training a machine learning agent. This task has a relatively manageable sample space for algorithmic approach, and hence, there is no need for machine learning. Moreover, the prediction task contains randomness such that the error is high even when the agent reaches optimal solution so there will be a lot of fluctuation of accuracy. However, this could help for mainly two reasons:

1. Ensuring that there exists a strategy before we started trying to come up with one
2. Approximate the accuracy that we can achieve with our guessing algorithm.

The code of a simple Recurrent Neural Network is done with Tensorflow 1.10 (I am pretty sure this RNN implementation I used here has already been deprecated in TF 2.0) and hyperparameter is as follow:

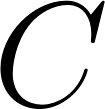
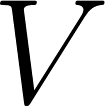
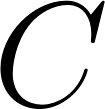
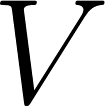
|  |
| --- |
| deck\_size = 52 embedding\_size = 128 **if** train:  iteration = 2000  batch\_size = 500 **else**:  iteration = 10000  batch\_size = 1 hidden\_size = 1024 layers = 3 learning\_rate = 1e-4  x = tf.placeholder(tf.int32, [batch\_size, deck\_size]) y = tf.placeholder(tf.int32, [batch\_size, deck\_size])  embedding = tf.Variable(tf.random\_normal([deck\_size+1, embedding\_size])) # 1 extra for the start token card\_embed = tf.nn.embedding\_lookup(embedding, x)  cell = rnn.MultiRNNCell([rnn.LSTMCell(hidden\_size) **for** \_ **in** range(layers)] + [rnn.LSTMCell(deck\_size)]) fc, state = tf.nn.dynamic\_rnn(cell, card\_embed, dtype = tf.float32)  Loss = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels = y, logits = fc)  out\_loss = tf.reduce\_mean(Loss) opt = tf.train.AdamOptimizer(learning\_rate).minimize(Loss) |

5. Card Guessing Algorithm

According to the analysis from the sections above, we developed a card guessing technique by constructing increasing sequences out of the shuffled deck.

5.1 Overhand Shuffle

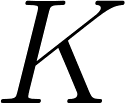
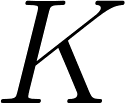
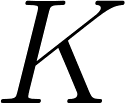
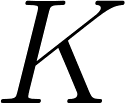
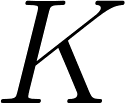
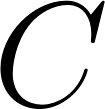
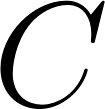
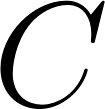
Below are the steps of card guessing algorithm for Overhand Shuffle:

1. If it is the first round, we guess 1.
2. Otherwise, we take the revealed card [](https://www.codecogs.com/eqnedit.php?latex=C%0) from the last turn as reference. Now we make our prediction as the smallest possible valid card [](https://www.codecogs.com/eqnedit.php?latex=V%0) such that:
   1. It has not been revealed yet.
   2. It is larger than [](https://www.codecogs.com/eqnedit.php?latex=C%0).
   3. It is in range of the numbers in the deck.
3. If no such [](https://www.codecogs.com/eqnedit.php?latex=V%0) exists, we guess randomly from the possible cards (i.e. the cards which have not been revealed yet).
4. Reveal the target card and compare it to our prediction. Record the results.

|  |
| --- |
| **if** self.style == "Overhand Shuffle":  cards = list(range(1, 53))  step = 0  guess = 0  **while** cards:  **if** step == 0: # Step 1.  guess = 1  **else**:  # Step 2.  i = 0  **while** i < len(cards):  **if** cards[i] > self.target[step - 1]:  guess = cards[i]  **break**  i += 1    # Step 3.  **if** i == len(cards):  guess = random.choice(cards)     **if** guess == self.target[step]:  correct += 1    guesses.append(guess)  cards.remove(self.target[step])    step += 1  **return** correct, guesses |

5.2 Riffle Shuffle

Below are the steps of card guessing algorithm for Riffle Shuffle:

1. In each round we construct a list of **Valid Increase Sequence** by taking the next number [](https://www.codecogs.com/eqnedit.php?latex=K%0) of each **Known Increase Sequence** by adding 1 to them.
   1. If [](https://www.codecogs.com/eqnedit.php?latex=K%0) is strictly greater than 52 or [](https://www.codecogs.com/eqnedit.php?latex=K%0) was already flipped out of the deck, we discard [](https://www.codecogs.com/eqnedit.php?latex=K%0).
   2. Otherwise, we append [](https://www.codecogs.com/eqnedit.php?latex=K%0) to the list of **Valid Increase Sequence**, and assign **Valid Increase Sequence** to the list of  **Known Increase Sequences**.
2. If there is no element in the list of **Known Increase Sequences**, we guess the first available card (i.e. 1 at the very beginning). Otherwise, we pick a random element from **Known Increase Sequences** as our guess.
3. Reveal the card we try to guess and increment the number of correct guesses if we guessed correctly. If the revealed card [](https://www.codecogs.com/eqnedit.php?latex=C%0) belongs to one of the **Known Increase Sequences**, we update the encountered number of that sequence to [](https://www.codecogs.com/eqnedit.php?latex=C%0). Otherwise, the card is the beginning of a new increase sequence, so we append [](https://www.codecogs.com/eqnedit.php?latex=C%0) to the list of **Known Increase Sequences**.

|  |
| --- |
| **if** self.style == "Riffle Shuffle":  cards = list(range(1, 53))  inc\_seq = []  step = 0  guess = 0  **while** cards:  # Step 1.  valid\_inc\_seq = []  **for** i **in** range(len(inc\_seq)):  **if** (inc\_seq[i]+1) < 53 **and** (inc\_seq[i]+1) **in** cards:  valid\_inc\_seq.append(inc\_seq[i]+1)    # Step 2.  **if** vaild\_inc\_seq:  guess = random.choice(valid\_inc\_seq)  **else**:   guess = cards[0]   # Step 3.  **if** guess == self.target[step]:  correct += 1   **if** self.target[step]-1 **in** inc\_seq:  inc\_seq\_ind = inc\_seq.index(self.target[step]-1)  inc\_seq[inc\_seq\_ind] = self.target[step]  **else**:  inc\_seq.append(self.target[step])   guesses.append(guess)  cards.remove(self.target[step])  step += 1   **return** correct, guesses |

6. Results of Experiments

The full Python code of shuffling and card guessing algorithm code be viewed and run here: <https://repl.it/@YuChao1/ESTR2002> (Since running the Machine Learning Agent with Tensorflow on Repl crashes the webpage, the code is not deposited online. You may refer to the previous session for the code implementation and the hyperparameters.)

Basic Information: The accuracy of Machine Learning Agent is calculated with 10,000 samples while the algorithm is calculated with 100,000 samples.

These experiments have 2 controls. The first one is a well shuffled deck of poker, which we achieve via random.shuffle( ) function provided by Python3 IDE.

Our algorithms for Overhand Shuffle and Riffle Shuffle both achieved average accuracy of 4.538 cards while our Machine Learning Agent got 4.515 cards correct on average.

The second control is an unshuffled deck of poker. Both algorithms and the ML Agent obtained a perfect prediction of 52 cards.

The following table shows our prediction accuracy of Overhand Shuffle. We sampled the result in an interval of 5 from 0 to 20 and compared it to that of Neural Network. We can observe that Neural Network seems to perform poorly relative to that of algorithm, this could be due to:

1. Unoptimized implementation and hyperparameters of our Recurrent Neural Network.
2. In this task, there could exist an optimal solution that is achievable with algorithm. However, gradient descent used in Neural Network will not be able to properly identify and halt at the optimal solution. Instead, it keeps on trying to make changes even after reaching optimal solution, which will instead result in worse accuracy.

|  |  |  |
| --- | --- | --- |
| \Agent  Overhand Shuffle (times) | Algorithm (cards)  (rounded to 4 d.p.) | Neural Network (cards) |
| 0 | 52.0 | 52.0 |
| 1 | 50.0027 | 48.0201 |
| 5 | 44.5481 | 41.2839 |
| 10 | 37.9937 | 33.5312 |
| 15 | 32.3651 | 28.5322 |
| 20 | 27.6372 | 23.3363 |

Table 1: Prediction Accuracy of Overhand Shuffle of Algorithm & Neural Network [from 0 to 20, interval 5]

We also sampled the accuracy of the algorithm from 100 to 445 Overhand Shuffles in the interval of 5. We discovered that with around 250 Overhand Shuffles (x-axis = 25 in Figure 1), the accuracy of prediction converges to 4.538 cards (indicated in dotted red lines in Figure 1), which is the accuracy of an uniformly random deck. The number is drastically lower than 4640 Shuffles stated in (2.3 Analysis). This could mean that the algorithm still has room for improvement.

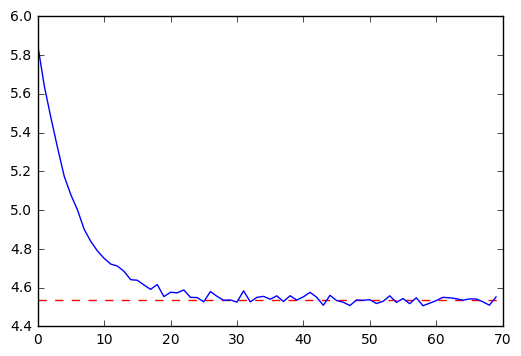


Figure 1:Prediction Accuracy of Overhand Shuffle of Algorithm [from 100 to 445, interval 5]

The following table shows our prediction accuracy of Riffle Shuffle. We sampled the result from 0 to 6 for algorithm & Neural Network and from 6 to 11, which includes the proposed mixing time of 8.55 shuffles (3.3 Analysis) of Riffle Shuffle, for algorithm only. The sampling of ML Agent stops at 6 due to the lack of time for training and testing (we trained separate model for different ways of shuffling to obtain maximum accuracy, i.e. RNN for riffle shuffle 5 times and RNN for riffle shuffle 6 times is trained completely independently). Two major findings can be derived from the following table:

1. With 8.55 shuffles (3.3 Analysis), the accuracy of prediction of the algorithm is in between 4.6497 and 4.5972 cards. To further diminish the performance of predictive algorithm to around 4.538 cards, we would recommend doing 13 Riffle Shuffles instead.
2. By comparing the result between algorithm and Neural Network prediction from 0 to 5 shuffles, we discovered that the Neural Network obtain a better prediction. This likely suggests that the trained Neural Network managed to deduce pattern other than the increasing sequences, which allows it to outperform the algorithm that relies on increasing sequences only. However, the exact pattern is unknown since we are unable to extract this information from the Neural Network model.

|  |  |  |
| --- | --- | --- |
| Riffle Shuffle (time) \ Agent | Algorithm (cards)  (rounded to 4 d.p.) | Neural Network (cards) |
| 0 | 52.0 | 52.0 |
| 1 | 27.0099 | 26.6748 |
| 2 | 14.8554 | 15.0923 |
| 3 | 9.1218 | 9.3266 |
| 4 | 6.5848 | 6.8347 |
| 5 | 5.4318 | 5.8590 |
| 6 | 4.9605 | 5.2284 |
| 7 | 4.7163 | N/A |
| 8 | 4.6497 | N/A |
| 9 | 4.5972 | N/A |
| 10 | 4.5564 | N/A |
| 11 | 4.5488 | N/A |
| 12 | 4.5408 | N/A |
| 13 | 4.5365 | N/A |

Table 2: Prediction Accuracy of Riffle Shuffle of Algorithm & Neural Network [from 0 to 13]

We verified the above finding(2) by visualizing the accuracy over time steps of 5 Riffle Shuffle. Since the player will make 52 guesses in the game, the x-axis in the following graph is ranged from 0 to 51. We noticed that the percentage of making correct guesses of Neural Network (green line) increases faster than that of algorithm (blue line) at around step 20 to 50.

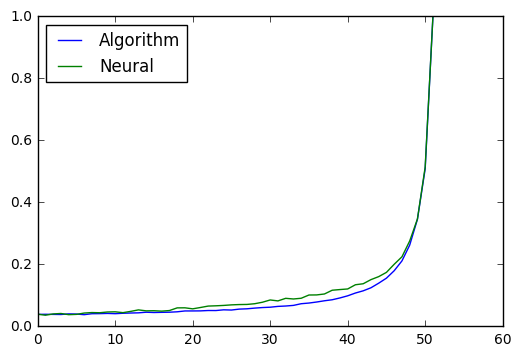


Figure 2: Accuracy over Time Steps of 5 Riffle Shuffle

**Data of Figure 2**

neural: array([0.0395, 0.0362, 0.0404, 0.0418, 0.038 , 0.0389, 0.0432, 0.0448,

0.0442, 0.0466, 0.0472, 0.0445, 0.0486, 0.0534, 0.0503, 0.0507,

0.0494, 0.0513, 0.0599, 0.0601, 0.0568, 0.0611, 0.0657, 0.0666,

0.0678, 0.0696, 0.0706, 0.071 , 0.0733, 0.0781, 0.0854, 0.0824,

0.0907, 0.0886, 0.0909, 0.1014, 0.1017, 0.1047, 0.1168, 0.119 ,

0.1209, 0.1345, 0.1377, 0.151 , 0.1607, 0.1744, 0.2005, 0.2246,

0.2756, 0.3466, 0.5143, 0.9997])

algorithm : array([0.0393, 0.0368, 0.0385, 0.0403, 0.04 , 0.0419, 0.0365, 0.0448,

0.0417, 0.0457, 0.0414, 0.0448, 0.0412, 0.0448, 0.047 , 0.0473,

0.0494, 0.0488, 0.0457, 0.0523, 0.0504, 0.0493, 0.05 , 0.0531,

0.0568, 0.054 , 0.057 , 0.06 , 0.0575, 0.0599, 0.062 , 0.0619,

0.0621, 0.0694, 0.0707, 0.0731, 0.0757, 0.0849, 0.0859, 0.0893,

0.1012, 0.1045, 0.1128, 0.1213, 0.1347, 0.1561, 0.1814, 0.2156,

0.2631, 0.3426, 0.5056, 1. ])

**7. Optimal Prediction Algorithm for Riffle Shuffle**

D. Bayer and P. Diaconis theorized the accuracy of a conjectural optimal prediction algorithm for Riffle Shuffle. The number of cards predicted correctly of said algorithm is much higher than the algorithm we simulated and our Neural Network prediction. However, such algorithm is only theorized by mathematical induction and the actual implementation has not been known yet.

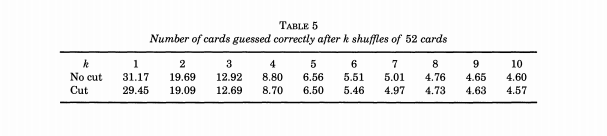


Figure 3: Table of accuracy of conjectural optimal prediction algorithm

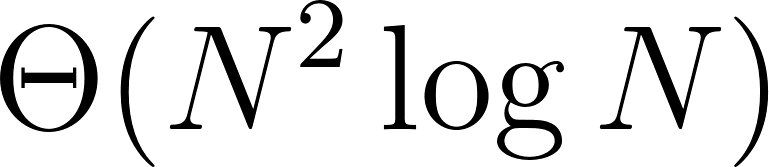
8. References

[1] E. Glibert, *Theory of shuffling*, 1955.

[2] D. Bayer and P. Diaconis, *Trailing the Dovetail Shuffle to its Lair*, 1992. (<https://projecteuclid.org/download/pdf_1/euclid.aoap/1177005705>)

[3] D. Aldous and P. Diaconis, *Shuffling Cards and Stopping Times*, 1986.

(<https://www.jstor.org/stable/2323590?seq=1#metadata_info_tab_contents>)

[4] J. Jonasson, *The Overhand Shuffle Mixes in* [**](https://www.codecogs.com/eqnedit.php?latex=%5CTheta(N%5E2%20%5Clog%20N)%0) *Steps*, 2006.

(<https://arxiv.org/pdf/math/0501401.pdf>)

[5] Numberphile (via YouTube), *The Best (and Worst) Ways to Shuffle Cards*, 2015.

(<https://youtu.be/AxJubaijQbI>)

[6] Robin Pemantle, Randomization Time for the Overhand Shuffle, 1988.

(<https://www.math.upenn.edu/~pemantle/papers/overhand.pdf>)