Logic Simulation Test

& ML AI Opponent Development

Drafted by:

Levi Holmstrom

Objective

- Understand the gameplay and logic provided within the documentation.
- Break-down components of game mechanics and build them in Python.
- Achieve a functioning program and perform testing. Analyse outputs.
- Use collected data to train an AI opponent using machine learning.
- Capture the model, and implement it into the program.
- Analyse results and sign off for next phase.

Approach

1. Configure the core logic into a headless client to allow looping of the game mechanics.
2. Construct a defined framework that allows easy manipulation of variables
3. Introduce a new variable each testing phase until desired results are achieved.
4. Capture and plot results.
5. Examine results and determine if expenditure of testing is required.

Environment Setup

- Language:
  - Python
    - Modules:
      - Pygame, Random, Enum, Time, TensorFlow, Keras, Scikit
- IDE:
  - PyCharm
- Database:
  - MS Excel
- Learning Resource:
  - GPT3.5/4, Coding Discord, Google Search, YouTube (IBM Data Analyst from Australia)

Core Game Logic

Please refer to blindeye_rules_rev2 for a more in-depth explanation of the game mechanics.

https://github.com/iMlearnDinG/Python-Training/blob/main/blindeye_RULES_rev2.pdf

Variables
(Static)

- Rank
- Suit

```
class Suit(Enum):
    Spades = 4
    Hearts = 3
    Diamonds = 2
    Clubs = 1

def __init__(self):
    self.cards = [self.Card(rank, suit) for rank in range(1, 14) for suit in self.Suit]
    self.discard_pile = []
```

Win Condition

- Player with the shortest count to the dealer rank wins
- If rank proximity is a tie, we use a static suit value to determine the winner.
- If rank proximity and suit value are both tied, a pair of dice is rolled for each player. The highest scoring dice roll determines the winner.

```
rank_diff1 = min(abs(player1_card.rank - dealer_card.rank), abs(13 - abs(player1_card.rank - dealer_card.rank)))
    rank_diff2 = min(abs(player2_card.rank - dealer_card.rank), abs(13 - abs(player2_card.rank - dealer_card.rank)))

    if rank_diff1 < rank_diff2:
        return "player1"
    elif rank_diff1 > rank_diff2:
        return "player2"
    else:
        if player1_card.suit.value > player2_card.suit.value:
            return "player1"
        elif player1_card.suit.value < player2_card.suit.value:
            return "player2"
        else:
```

Tie Determination

- Dice Roll
- Value Comparison (Higher Wins)

```
def roll_dice():
    return random.randint(1, 6)

print("Tie! Rolling dice...")
player1_dice = roll_dice()
player2_dice = roll_dice()
print("Player 1 rolled a", player1_dice)
print("Player 2 rolled a", player2_dice)

if player1_dice > player2_dice:
    score.update_score("player1")
elif player1_dice < player2_dice:
    score.update_score("player2")
```

Development

Main Program

Class definitions

. Cards .

Rank

Suit

```
class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __str__(self):
        return f"{self.rank} {self.suit.name}"
```

. Deck .

Shuffle

Deal

Discard

```
class Deck:
    def __init__(self):
        self.cards = [Card(rank, suit) for rank in
range(1, 14) for suit in Suit]
        self.discard_pile = []

    def shuffle(self):
        random.seed(time.time())
        random.shuffle(self.cards)

    def deal(self):
        if not self.cards:
            self.cards, self.discard_pile =
self.discard_pile, self.cards
            self.shuffle()
        return self.cards.pop()

    def discard(self, card):
        self.discard_pile.append(card)
```

. Player/Dealer .

Hand

Score Track

```
class Player:
    def __init__(self, name):
        self.name = name
        self.score = 0
        self.hand = []

    def add_point(self):
        self.score += 1
```

. Assets .

Implementation @ later date

```
class Assets:
    def __init__(self):
        pass

    def draw(self):
        pass
```

. Score .

Score Add

Score Update

Score Display

```
class Score:
    def __init__(self):
        self.players = {"player1": 0, "player2": 0}

    def update_score(self, player):
        self.players[player] += 1

    def get_score(self, player):
        return self.players[player]
```

Function definitions

. Proximity .

```
def rank_proximity(player_card, dealer_card):
    return abs(player_card.rank - dealer_card.rank) % 13


def suit_proximity(player_card, dealer_card):
    return abs(player_card.suit.value -
dealer_card.suit.value)
```

. Comparison .

```
def compare_cards(player1_card, player2_card,
dealer_card):
    rank_diff1 = min(abs(player1_card.rank -
dealer_card.rank), abs(13 - abs(player1_card.rank -
dealer_card.rank)))
    rank_diff2 = min(abs(player2_card.rank -
dealer_card.rank), abs(13 - abs(player2_card.rank -
dealer_card.rank)))

    if rank_diff1 < rank_diff2:
        return "player1"
    elif rank_diff1 > rank_diff2:
        return "player2"
    else:
        if player1_card.suit.value >
player2_card.suit.value:
            return "player1"
        elif player1_card.suit.value <
player2_card.suit.value:
            return "player2"
        else:
            return "tie"
```

. Print Player Card Info .

```
def print_cards(players, dealer):
    print("Player 1 cards:")
    for card in players[0].hand:
        print(card)
    print("\nPlayer 2 cards:")
    for card in players[1].hand:
        print(card)
    print("\nDealer cards:")
    for card in dealer.hand:
        print(card)
```

. Print Dice-Roll Results .

```
def roll_dice():
    return random.randint(1, 6)
```

Main Initialization

```
def main():
  pygame.init()
  assets = Assets()
  players = [Player("player1"), Player("player2")]
  dealer = Player("dealer")
  score = Score()

  # Initialize and deal cards for the first game
  deck = Deck()
  deck.shuffle()
  for _ in range(5):
    for player in players:
      player.hand.append(deck.deal())
    dealer.hand.append(deck.deal())

  print_cards(players, dealer)

  playing = True
  while playing:
    assets.draw()

    for i in range(5):
      player1_card = players[0].hand[i]
      player2_card = players[1].hand[i]
      dealer_card = dealer.hand[i]

      winner = compare_cards(player1_card, player2_card, dealer_card)

      if winner == "player1":
        score.update_score("player1")
      elif winner == "player2":
        score.update_score("player2")
      else:
        print("Tie! Rolling dice...")
```

Output

First Output	Output /w Tie
Player 1 cards:	Player 1 cards:
1 Diamonds	10 Spades
10 Clubs	3 Spades
13 Clubs	9 Clubs
11 Spades	4 Hearts
7 Spades	8 Hearts
Player 2 cards:	Player 2 cards:
12 Diamonds	5 Spades
4 Spades	7 Clubs
3 Clubs	12 Hearts
12 Hearts	13 Clubs
9 Diamonds	1 Hearts
Dealer cards:	Dealer cards:
9 Spades	11 Hearts
2 Spades	4 Clubs
9 Clubs	7 Diamonds
7 Clubs	8 Diamonds
10 Diamonds	11 Diamonds
Current scores:	Tie! Rolling dice...
Player 1: 2	Player 1 rolled a 4
Player 2: 3	Player 2 rolled a 6

First output shows promising results. The player scores align with the implemented logic. A complete turn with no errors has been achieved.

Testing

We will now test the logic to retrieve the following statistics:

- **Baseline / Game Balance**
- **Win Acceleration**
- **Score Distribution**
- **Outcome Manipulation**
- **Introduce Variables**

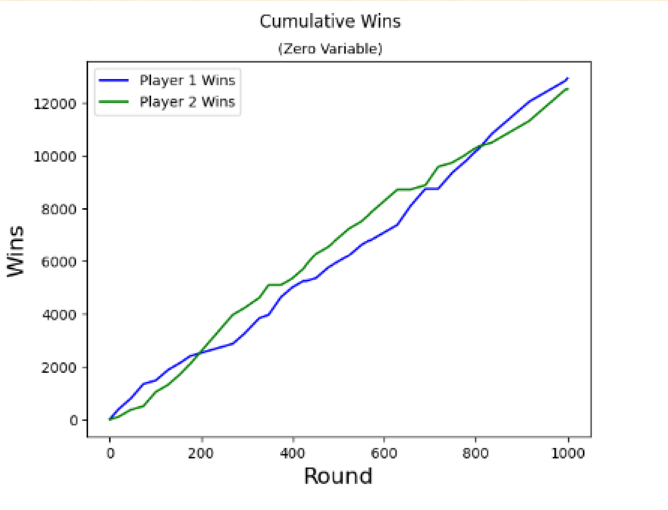
Baseline / Balance

Player 1- & 2- Win Condition Totals.

3 Cycle Counts to determine dataset accuracy.

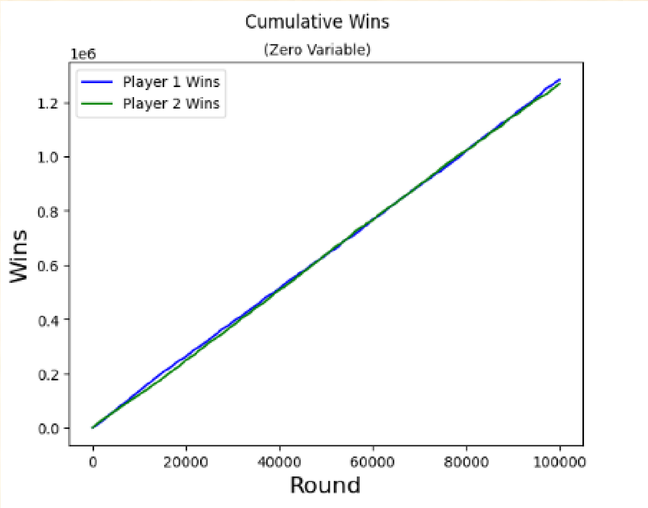
Game variables - INACTIVE

Rounds: 1,000



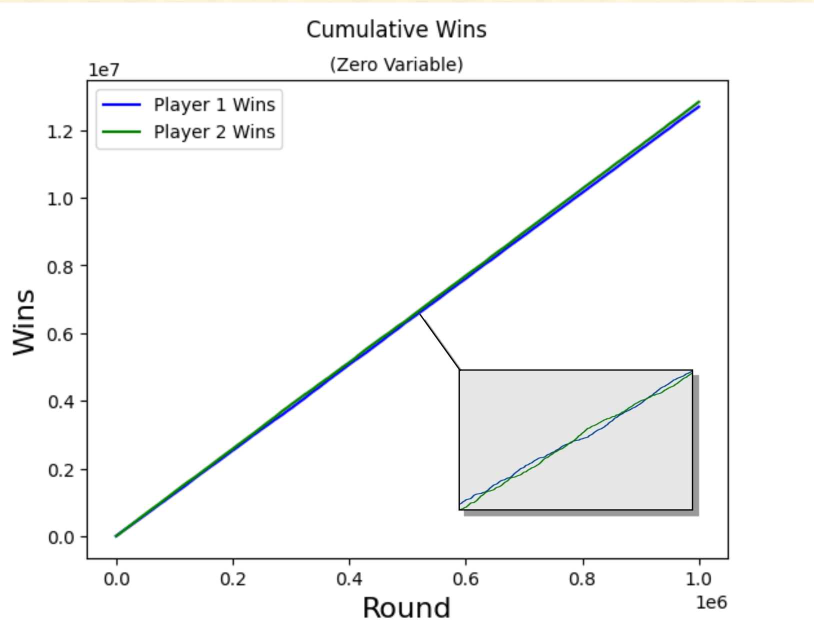
Total scores:
Player 1 – 12930 (50.80%)
Player 2 – 12519 (49.18%)
Ties – 6 (0.02%)

Rounds: 100,000



Total scores:
Player 1 – 1282819 (50.02%)
Player 2 – 1268242 (49.45%)
Ties – 13649 (0.53%)

Rounds: 1,000,000



1 x 10^6 provides a thoroughly accurate baseline for preliminary results, with each player having almost a 50/50 chance of winning.

Also observed is the geometric form the graph lines take; it almost resembles DNA strands… interesting!

More research into this by-result is warranted.

Total scores:
Player 1 – 12681906 (49.48%)
Player 2 – 12823843 (50.03%)
Ties – 124036 (0.48%)

Testing

continued...

Win Acceleration

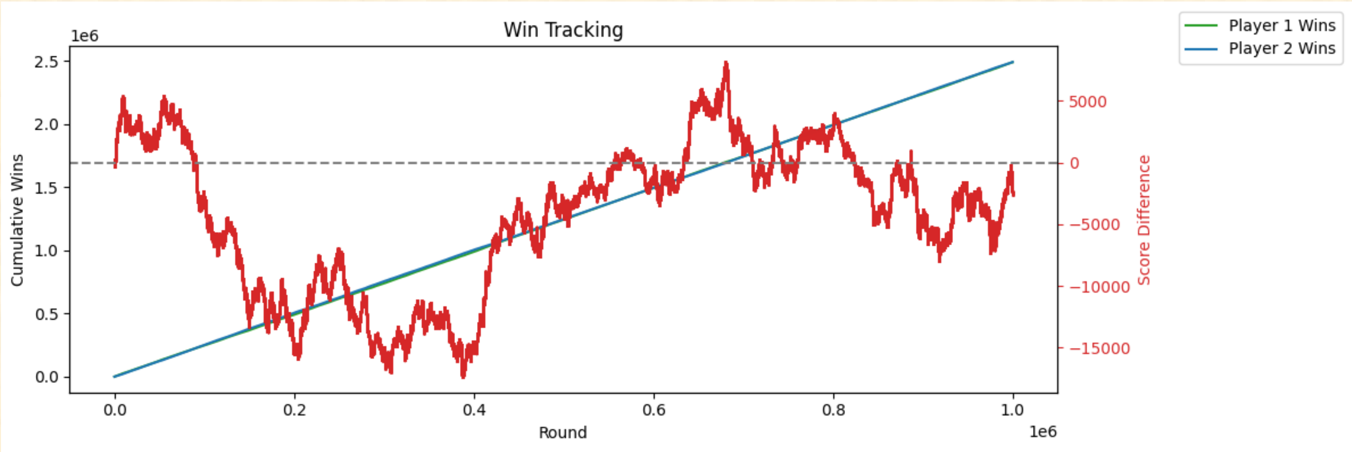
Visualise the number of times players are winning on the timeline.

+0 = Player 1 (*above line*)
-0 = Player 2 (*below line*)

Player 2 *has a significant lead at the start of game and peaks out at around 16k score ahead at 400k games.*

Player 1 *has a significant lead from 400k to 700k peeking out at 7k ahead in score.*

The remaining rounds *play out evenly, with the score difference returning to the zero line at the end of the game, indicating a fair chance for each player*



Score Distribution

Visualise the score value designation for each round.

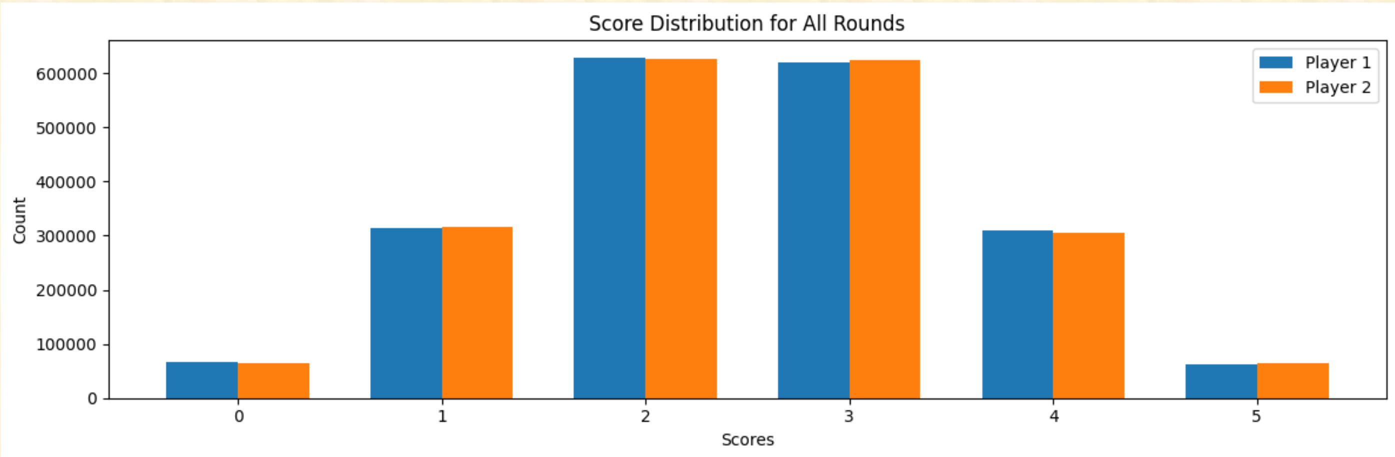
Help determine common vs rare scores

Testing *reports even score distribution for both players. This is exceptionally handy for setting a preliminary baseline betting ratio.*

0 & 5 – 5x

1 & 4 – 2.5x

2 & 3 – 1.5x



Testing

continued...

Outcome Manipulation

Bias penetration prevention and detection threshold.

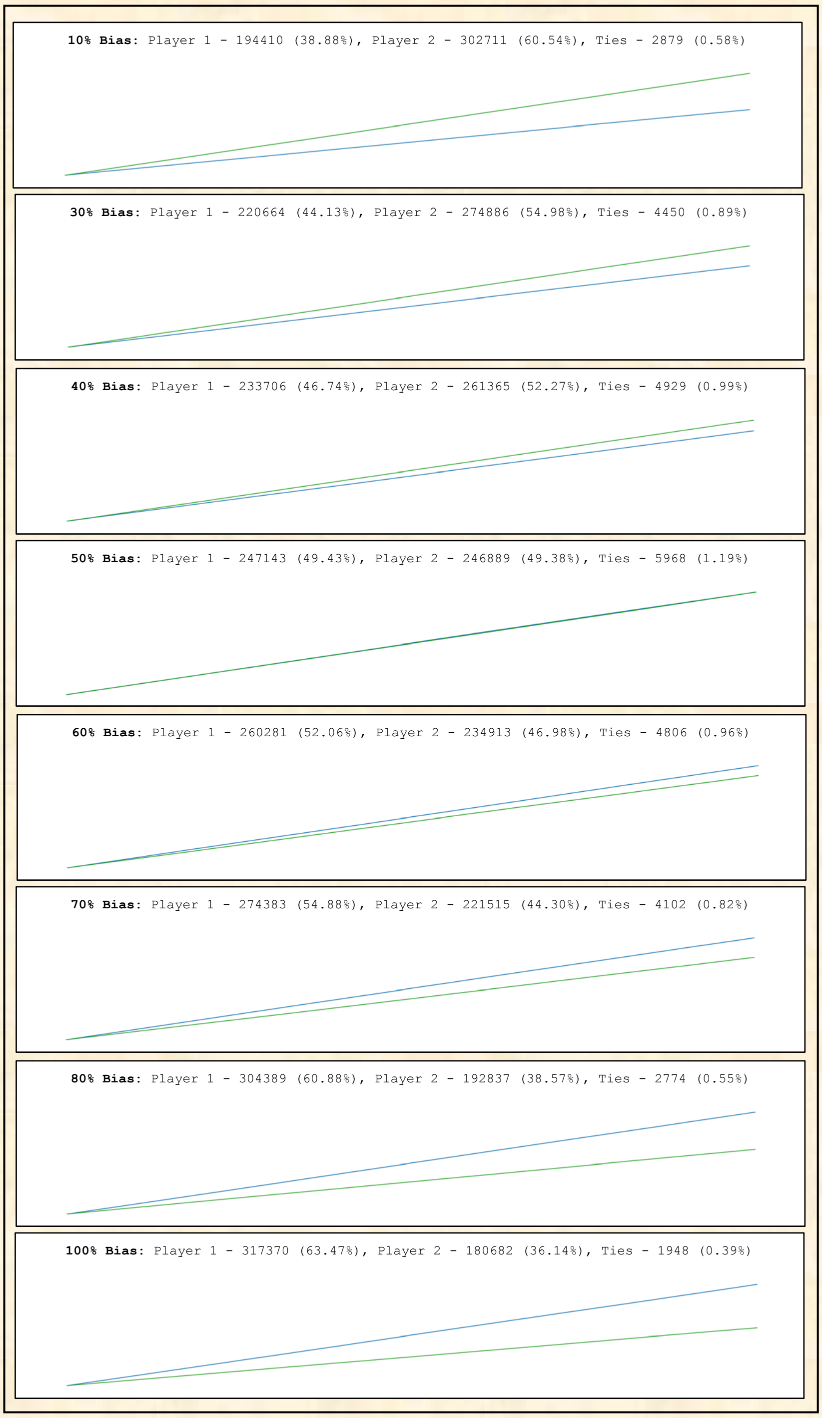
100,000 Rounds x 8

Artificial Variables:

Player 1 – More odds of receiving higher suit from dealer. (Spades)

Player 2 – More Odds of receiving lower rank suits from dealer. (Clubs)

Bias Testing Results



Manipulation Definition

```
def deal_biased_cards(deck,
bias_factor=1):
    high_ranked_cards = [card for card
in deck if card[1] > 6]
    low_ranked_cards = [card for card
in deck if card[1] <= 6]

    num_high_ranked_cards = int(5 *
bias_factor)
    num_low_ranked_cards = 5 -
num_high_ranked_cards

    random.shuffle(high_ranked_cards)
    random.shuffle(low_ranked_cards)

    biased_hand_high =
high_ranked_cards[:num_high_ranked_car
ds]
    biased_hand_low =
low_ranked_cards[:num_low_ranked_cards
]
    random.shuffle(biased_hand_high)
    random.shuffle(biased_hand_low)

    for card in biased_hand_high +
biased_hand_low:
        deck.remove(card)

    return biased_hand_high +
biased_hand_low
```

Manipulated Shuffle/Deal

```
# Shuffle deck and deal cards
    deck = [(f"{k} of {s}",
card_values[k], suit_values[s])
for k in card_values for s in
suit_values for _ in
range(int(k))]
    random.shuffle(deck)
    player1_cards =
deal_biased_cards(deck,
bias_factor=0.5)  # You can
adjust the bias_factor to
increase or decrease the bias
    player2_cards =
deal_biased_cards(deck,
bias_factor=1 - 0.5)  # Subtract
the bias_factor from 1 to create
an inverse bias for player 2
    dealer_cards = [deck.pop()
for _ in range(5)]
```

***The results show** that a bias penetration is possible with a code injection.*

This could be due to the simple nature of the Random Algorithm being used.

A more advanced algorithm is warranted for testing to see if a reduction in baseline deviation is achieved.

Testing

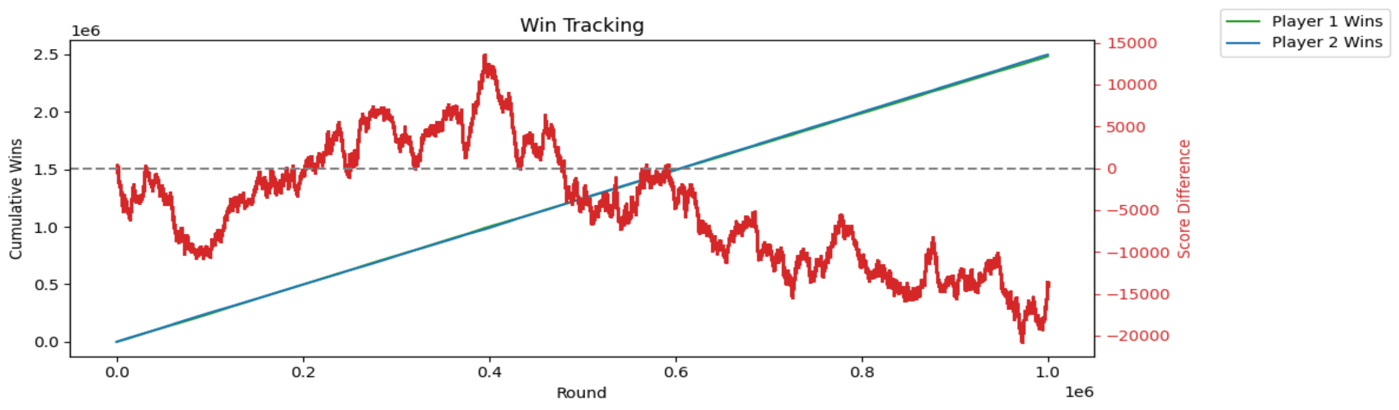
continued...

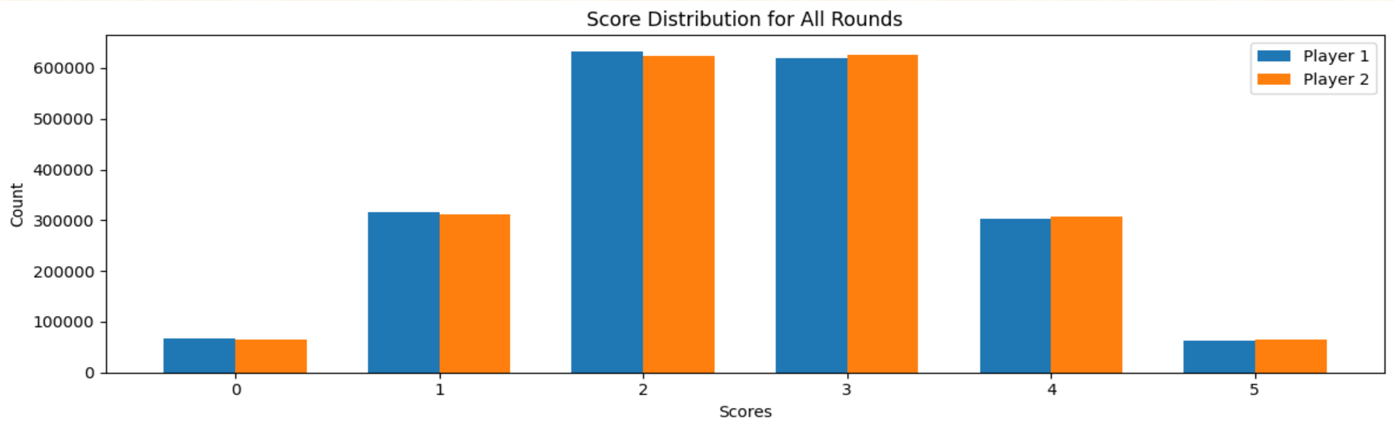
Introduce Variable

Game Mechanic Variable - ACTIVE

5th card swapped with new card to simulate Blind Eye.

***No noticeable deviation** from the initial game balance results. Reinforces game mechanics are performing as intended.*





Testing

Conclusion

From the testing results, we can determine the following:

- Game is balanced
- Variables cause minimal impact
- Bias can be tuned for targeted outcomes
- Even Score Distribution

Machine Learning

for AI opponent decision making

Training will now commence of the ML Model. This model will be used as the brain for the CPU opponent to make its decisions.

We will use a ***Feedforward Neural Network*** machine learning architecture as this has been pre-determined as the strongest algorithm for the task. The training will stop after ***3 consecutive failures*** at gaining accuracy during the training epoch. *(Max training limit: 1000 Epochs)*

```
def create_model(neurons=1, optimizer='adam', learning_rate=0.001):
    print(f"Creating model with {neurons} neurons")
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(neurons, input_shape=input_shape, activation='relu'),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dense(5, activation='sigmoid')  # 5 output neurons for 5 
columns
    ])

    # Set the optimizer with the specified learning rate
    optimizer = optimizer.lower()
    if optimizer == 'adam':
        optimizer = Adam(learning_rate=learning_rate)

    model.compile(optimizer=optimizer,
                  loss='binary_crossentropy',  # use binary_crossentropy for multi-
label problems
                  metrics=['accuracy'])
    return model


model = KerasClassifier(build_fn=create_model, epochs=1000, verbose=1,
                        callbacks=[EarlyStopping(monitor='accuracy', patience=3, 
mode='max'), PrintParams()])
```

A grid-search will be initiated at the beginning to iterate through the following hyper-parameters to find the training method with the ***highest mean_accuracy***.

```
param_grid = {
    'neurons': [64, 128, 256],
    'optimizer': ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam'],
    'batch_size': [32, 64, 128]
}
```

The script is using the previous 1,000,000 games data to try and predict the winner using the recorded outcomes. *(Example results below)*

P1_Card_1	P1_Card_2	P1_Card_3	P1_Card_4	P1_Card_5	P2_Card_1	P2_Card_2	P2_Card_3	P2_Card_4	P2_Card_5	DLR_Card_1	DLR_Card_2	DLR_Card_3	DLR_Card_4	DLR_Card_5	Winner_Column_1	Winner_Column_2	Winner_Column_3	Winner_Column_4	Winner_Column_5	P1_Score	P2_Score
12 Spades	8 Clubs	4 Diamonds	8 Hearts	8 Spades	13 Diamonds	1 Diamonds	11 Hearts	1 Clubs	10 Diamonds	7 Spades	11 Clubs	3 Clubs	12 Hearts	6 Hearts	player1	player2	player1	player2	player1	3	2
7 Spades	10 Spades	6 Spades	9 Clubs	10 Diamonds	4 Clubs	2 Spades	13 Clubs	5 Hearts	9 Diamonds	1 Clubs	4 Diamonds	11 Diamonds	12 Spades	8 Hearts	player2	player2	player2	player1	player2	1	4
10 Spades	5 Spades	7 Clubs	1 Spades	13 Hearts	13 Spades	13 Clubs	7 Diamonds	3 Spades	4 Diamonds	8 Hearts	6 Spades	12 Clubs	3 Clubs	11 Diamonds	player1	player1	player2	player2	player1	3	2
9 Spades	3 Clubs	12 Hearts	8 Hearts	7 Clubs	7 Spades	7 Diamonds	8 Clubs	2 Diamonds	13 Spades	12 Clubs	1 Clubs	1 Hearts	2 Spades	8 Diamonds	player1	player1	player1	player2	player1	4	1
7 Clubs	8 Spades	1 Diamonds	6 Diamonds	3 Spades	11 Clubs	2 Hearts	6 Hearts	9 Hearts	1 Clubs	4 Diamonds	12 Hearts	3 Diamonds	2 Clubs	10 Diamonds	player1	player2	player1	player1	player2	3	2

Full dataset available at :
https://drive.google.com/file/d/1_xExpMcBqRhppglQMtunkakLWcgl83iD/view?usp=drive_link

Machine Learning

Results of Grid Search

A total of 63 results were captured. The top 10 have been displayed below.

The highest-ranking parameters will be used to train the baseline model.

Batch_size: 128
Neurons: 128
Optimizer: SGD

grid_search_results_FINAL.csv

**Full grid search results available in the attached excel file.*

batch_size	neurons	optimizer	split0_test_score	split1_test_score	mean_test_score	std_test_score	rank	mean_accuracy	mean_loss
128	128	SGD	0.632289	0.5012661	0.5667777	0.0655116	***1***	0.56677779	-0.566777
64	256	RMSprop	0.535520	0.5842720	0.5598960	0.0243759	***2***	0.55989605	-0.559896
64	64	RMSprop	0.594589	0.5110514	0.5528202	0.0417688	***3***	0.55282026	-0.552820
128	128	RMSprop	0.618187	0.4789873	0.5485874	0.0696001	***4***	0.54858744	-0.548587
128	128	Adam	0.426361	0.6330417	0.5297017	0.1033400	***5***	0.52970172	-0.529701
32	256	RMSprop	0.495419	0.5408032	0.5181113	0.0226919	***6***	0.51811133	-0.518111
128	256	SGD	0.510376	0.5061134	0.5082449	0.0021314	***7***	0.50824490	-0.508244
64	64	Adam	0.503904	0.5102760	0.5070901	0.0031859	***8***	0.50709015	-0.507090
64	256	Nadam	0.443850	0.5615533	0.5027021	0.0588512	***9***	0.50270211	-0.502702
64	64	Nadam	0.5454370	0.4531974	0.4993173	0.0461198	***10***	0.49931737	-0.499317

Model Generation

The #1 ranking parameter configuration has been used in a separate training session to generate the model file. We can now import this file into the gameplay logic.

ai_best_model.h5

Model Generation

Training Details

Before we import our model, lets see how the model was trained.

The machine learning algorithm has read access to 15 cards each round. (Player 1, Player 2, Dealer)

Using the game logic available, it will try to predict which player has the winning hand.

The results of the 1,000,000 games played were converted into integer format.
Ties were not including in the models training, so the total number of trained games was ultimately ***909,280***.

0 – Loss
1 – Win

The first error in prediction appeared in **Game 72**

	A	B	C	D	E	F	G	H	I	J
1	Pred_Column_1	Pred_Column_2	Pred_Column_3	Pred_Column_4	Pred_Column_5	Actual_Column_1	Actual_Column_2	Actual_Column_3	Actual_Column_4	Actual_Column_5
68	0	0	0	1	1	0	0	0	1	1
69	1	1	1	1	1	1	1	1	1	1
70	0	0	0	1	1	0	0	0	1	1
71	1	1	1	1	1	1	1	1	1	1
72	1	0	1	1	1	0	0	1	1	1
73	0	1	1	0	0	0	1	1	0	0
74	1	1	0	0	0	1	1	0	0	0
75	0	0	0	1	0	0	0	0	1	0
76	1	1	0	1	0	1	1	0	1	0
77	1	1	1	0	0	1	1	1	0	0
78	0	0	0	1	1	0	0	0	1	1

**Pred_Column_X = Predicted Outcome*
**Actual_Column_X = Actual Outcome*

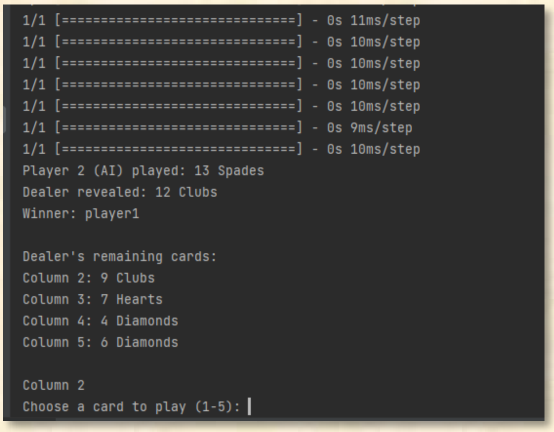
In total of the 909,280 games played, only 3298 were incorrectly predicted.

Model Test

First gameplay result:

Code Available:
https://github.com/iMlearnDinG/tensorflow-sandbox/blob/main/card_game_prediction/3.%20ai_opponent_test/ai_opponent_gameplay.py

Player 1 cards:
9 Diamonds
8 Spades
13 Clubs
2 Spades
1 Diamonds
Player 2 cards:
13 Spades
1 Hearts
8 Hearts
4 Spades
3 Diamonds
Dealer cards:
12 Clubs
9 Clubs
7 Hearts
4 Diamonds
6 Diamonds
Round 1
Choose a card to play (1-5): 3
Player 1 played: **13 Clubs**
1/1 [==============================] - 0s 10ms/step
Player 2 (AI) played: **13 Spades**
Dealer revealed: 12 Clubs
Winner: player1
Dealer's remaining cards:
Column 2: 9 Clubs
Column 3: 7 Hearts
Column 4: 4 Diamonds
Column 5: 6 Diamonds



```
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 10ms/step
1/1 [==============================] - 0s 10ms/step
1/1 [==============================] - 0s 10ms/step
1/1 [==============================] - 0s 10ms/step
1/1 [==============================] - 0s 9ms/step
1/1 [==============================] - 0s 10ms/step
Player 2 (AI) played: 13 Spades
Dealer revealed: 12 Clubs
Winner: player1

Dealer's remaining cards:
Column 2: 9 Clubs
Column 3: 7 Hearts
Column 4: 4 Diamonds
Column 5: 6 Diamonds

Column 2
Choose a card to play (1-5):
```

Round 2
Choose a card to play (1-5): 1
Player 1 played: 9 Diamonds
1/1 [==============================] - 0s 10ms/step
Player 2 (AI) played: 1 Hearts
Dealer revealed: 9 Clubs
Winner: player1
Dealer's remaining cards:
Column 2: 7 Hearts
Column 3: 4 Diamonds
Column 4: 6 Diamonds
Round 3
Choose a card to play (1-5): 1
Player 1 played: 8 Spades
1/1 [==============================] - 0s 10ms/step
Player 2 (AI) played: 8 Hearts
Dealer revealed: 7 Hearts
Winner: player2
Dealer's remaining cards:
Column 2: 4 Diamonds
Column 3: 6 Diamonds
Round 4
Choose a card to play (1-5): 1
Player 1 played: 2 Spades
1/1 [==============================] - 0s 10ms/step
Player 2 (AI) played: 4 Spades
Dealer revealed: 4 Diamonds
Winner: player2
Dealer's remaining cards:
Column 2: 6 Diamonds
Round 5
Choose a card to play (1-5): 1
Player 1 played: 1 Diamonds
1/1 [==============================] - 0s 10ms/step
Player 2 (AI) played: 3 Diamonds
Dealer revealed: 6 Diamonds
Winner: player2
Final scores:
Player 1: 2
Player 2: 3

The AI opponent *wins the first round! The selections it made seem to be healthy.*

Next Phase

Multiplayer Design

A mixture of Java, HTML & CSS will be used to construct a server and client.

Requirements:

Accessible via URL
Accounts & Scores
2 Player Lobbies
Real-time player interactions
Rewards or Betting system

Wrap up

We have successfully taken the provided concept in the Blindeye_Rules.pdf documentation and translated it into a functional program with interactive gameplay.

- Understand the gameplay and logic provided within the documentation. ✓
- Break-down components of game mechanics and build them in Python. ✓
- Achieve a functioning program and perform testing. Analyse outputs. ✓
- Use collected data to train an AI opponent using machine learning. ✓
- Capture the model, and implement it into the program. ✓
- Analyse results and sign off for next phase. ✓