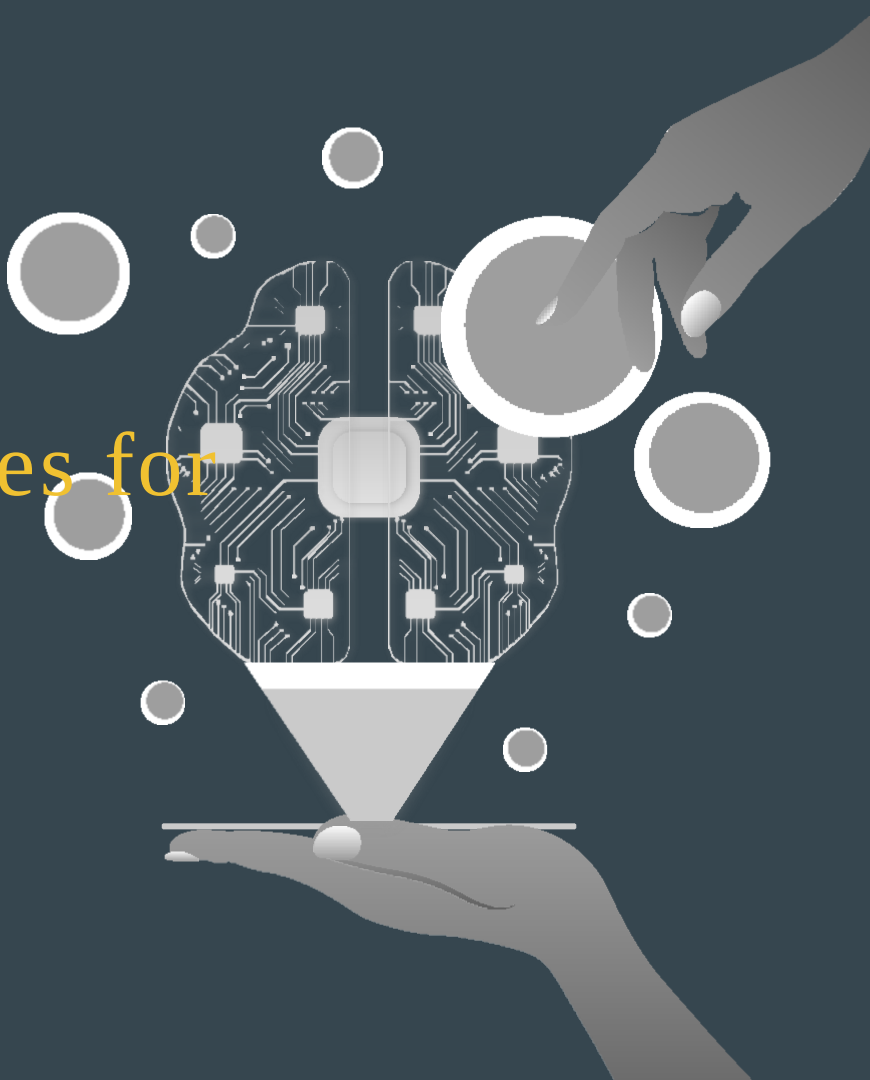
Evaluation of AI Strategies for Blackjack



Team 5
111950020 李欣穎
109550072 何嘉娫
109550202 白詩愷
110550092 林啟堯

PROBLEM STATEMENT

Blackjack is one of the most classic card games in the casino. Since this game has remained popular in casinos for such a long time, it is evident that the dealer usually has a higher winning probability than the player. This observation sparks our curiosity: Are there any strategies that can help players perform better in this game?

Introdution

The goal is to implement and test various AI agent to observe the winning rate for players under different methods. The strategies we investigate include, Random selection, Basic Strategy, Expectimax, Random Forest Classifier (RFC), and NeuroEvolution of Augmenting Topologies (NEAT).Through these experiments, we aim to answer the following questions: How does the player's win rate vary under different agents? Does the different number of decks affect the win rate? Are the casino's rules inherently designed to favor the dealer?

Objectives

1 Analyze two published ML-based Blackjack strategies

2 Implement a minimax (Expectimax) based strategy

3 Implement an AI agent and compare the methods

4 Identify the most effective AI approach

Timeline_

1. Justin Bodnar’s AI Model
2. Mason Lee Model
3. Our Contribution
4. Random and Basic Strategy
5. Reflect Agent Classifier (RFC)
6. Baseline-NeuroEvolution of Augmenting Topologies (NEAT)
7. Expectimax
8. Result and Analysis

01

Justin Bodnar Model

Goal of the Model

The goal of Bodnar's model is to train network that mimics a basic blackjack strategy that has:

- Input: Information regarding the current game state (e.g., player's hand value)
- Output: Decision to *hit* or *stay*.

Model Architecture

The model proposed uses Monte Carlo simulations of Blackjack games to create training data.

The model runs on a neural network:

- Level 1 →Player and hand value
- Level 2 →Player hand + dealer's face-up card
- Level 3 →All card seen

Mason Lee Model

Goal of the Model

The goal of this project is to train an AI agent to play Blackjack by evolving its decision-making strategy using NEAT (NeuroEvolution of Augmenting Topologies).

input: Current game state (player's hand, dealer's visible card, usable ace info, etc.)
Output: AI decides action: hit, stand, double

Model Architecture

Evolve neural networks based on fitness (win rate, performance)

1 →Initialize game
2 →Player decision making
3 →Dealer logic
4 →Determine result

Our Contribution;

- Implement Agent/ Training /Compare Win Rate of multiple AI strategies

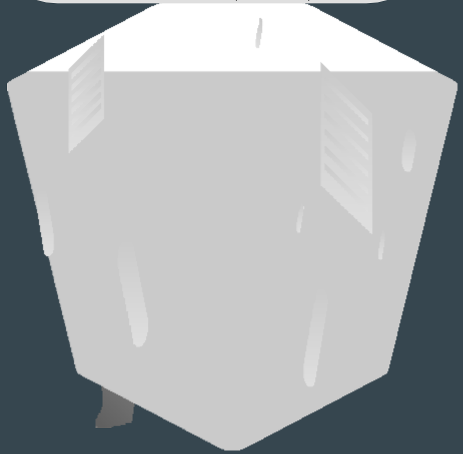
  (Random, Basic, RFC, NEAT)

- Implement these AI agents into an application about Blackjack

AI Strategies;

Random and Basic Strategy

Reflect Agent Classifier(RFC)



NeuroEvolution of Augmenting Topologies (NEAT)

Baseline-Random and Basic Strategy

The Random Agent makes decisions purely based on randomness. At each decision point, the agent randomly chooses between two actions: Hit/Stand

The Basic Strategy Agent strictly follows the static Blackjack decision table from the paper.

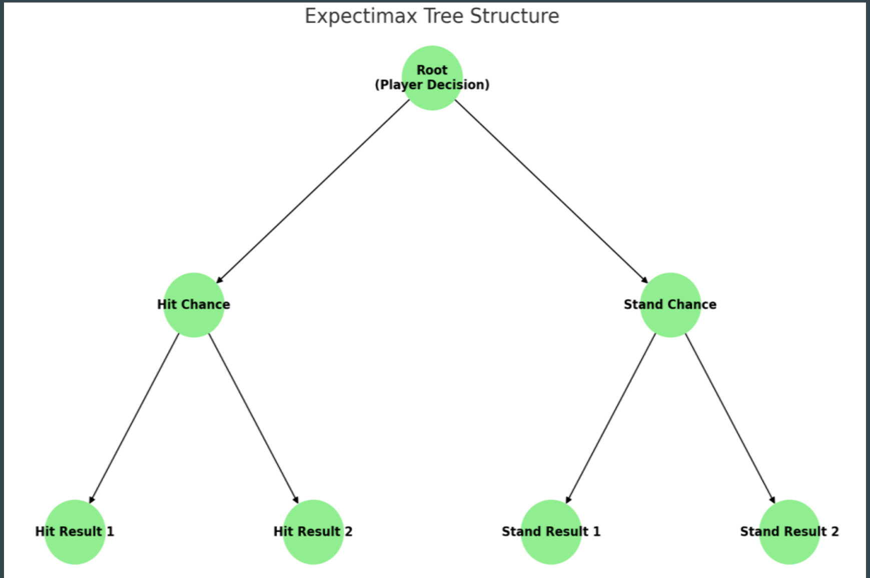
B.2. **One Up-Card.** The following table give the basic strategy for the one up-card variation played using one deck.

Player \ Dealer	2	3	4	5	6	7	8	9	10	11
Hard 4 - 12	H	H	H	H	H	H	H	H	H	H
Hard 13	H	H	H	H	H*	H	H	H	H	H
Hard 14	S*	H*	H*	S	S	H	H	H	H	H
Hard 15	S	S	S	S	S	H	H	H	H	H
Hard 16	S	S	S	S	S	H	H	H*	H*	S
Hard 17 - 20	S	S	S	S	S	S	S	S	S	S
Soft 12 - 16	H	H	H	H	H	H	H	H	H	H
Soft 17	S	S	S	H	H	H	H	H	H	H
Soft 18	S	S	S	S	S	S	S	H	S	S
Soft 18 - 20	S	S	S	S	S	S	S	S	S	S

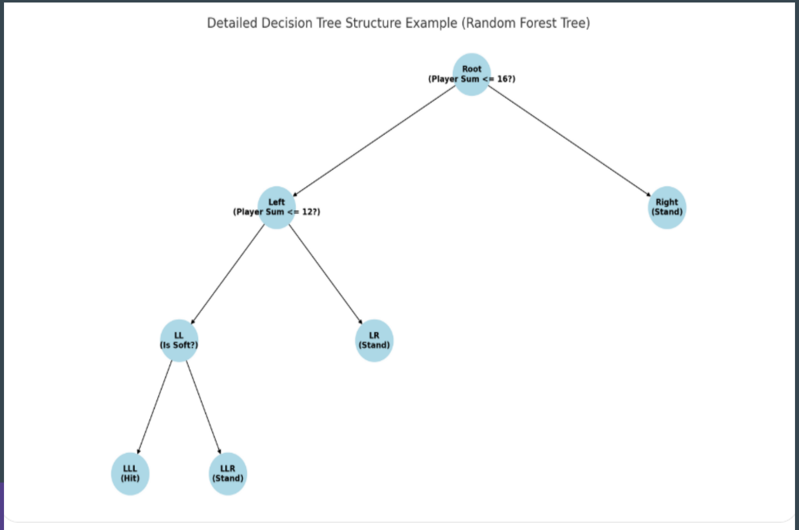
TABLE 7. Basic strategy for the one up-card variation (single deck). The * signifies that the optimal decision depends on the specific layout given this hand total.

Baseline-Expectimax

- **Build the Game Tree**
  Max nodes for player's choices (Hit/Stand)
  Chance nodes for random events (card draws)
- **Evaluate Leaf Nodes**
  Calculate terminal state values (win, lose, draw scores).
- **Backpropagate Values**
  Max nodes: Choose the **maximum value**.
  Chance nodes: Compute the **expected value** (weighted average of outcomes).
- **Choose Optimal Action**
  At the root, pick the action with the highest expected value.



Baseline-Random Forest Classifier (RFC)



1. Bootstrap Sampling: Randomly sample the training data with replacement to create a new dataset for each tree.
2. Feature Subset Selection: At each node split, randomly select a subset of features.
3. Tree Construction: Grow each decision tree to the maximum depth or until a stopping condition is met
4. Ensemble Prediction: For a new input, get predictions from all trees and use majority voting for classification.
5. Final Output: Return the majority class (classification) or average prediction (regression).

Baseline-NeuroEvolution of Augmenting Topologies (NEAT)

Step	Action
Initialize	Create random, simple networks
Fitness Eval	Run network → get fitness score
Speciation	Group by genetic distance (innovation #)
Fitness Sharing	Divide fitness within species
Select Parents	Based on adjusted fitness
Crossover	Align by innovation #, combine genes
Mutation	Adjust weights, add nodes/connections
New Generation	Offspring + elite genomes

Main Approach-AI Agent and Training Process

- Random Agent

  At each decision step, randomly selects Hit or Stand.

  No learning, no logic.

- Basic Strategy Agent

  Follows Blackjack basic strategy table, derived from the reference paper.

  Rules include:

Player \ Dealer	2	3	4	5	6	7	8	9	10	11
Hard 4 - 12	H	H	H	H	H	H	H	H	H	H
Hard 13	H	H	H	H	H*	H	H	H	H	H
Hard 14	S*	H*	H*	S	S	H	H	H	H	H
Hard 15	S	S	S	S	S	H	H	H	H	H
Hard 16	S	S	S	S	S	H	H	H*	H*	S
Hard 17 - 20	S	S	S	S	S	S	S	S	S	S
Soft 12 - 16	H	H	H	H	H	H	H	H	H	H
Soft 17	S	S	S	H	H	H	H	H	H	H
Soft 18	S	S	S	S	S	S	S	H	S	S
Soft 18 - 20	S	S	S	S	S	S	S	S	S	S

```python
class BasicStrategyAgent:
    def act(self, player_sum, dealer_card, is_soft, num_cards):
        if is_soft:
            if 12 <= player_sum <= 16:
                return 'hit'
            elif player_sum == 17:
                return 'stand'
            elif player_sum == 18:
                if dealer in [2, 7, 8]:
                    return 'stand'
                else:
                    return 'hit'
            elif player_sum >= 19:
                return 'stand'
            else:
                return 'hit'
        else:
            if 4 <= player_sum <= 8:
                return 'hit'
            elif player_sum == 9:
                if dealer in [3,4,5,6]:
                    return 'hit'
                else:
                    return 'hit'
            elif player_sum == 10:
```

- Purely rule-based, no learning.

Main Approach - Environment Code

- Deck Generation and Shuffling (reset_deck)
  Generates multiple decks of standard 52-card playing cards (1=A, 11=J, 12=Q, 13=K) and shuffles them randomly.
- Game Initialization (reset)
  At the start of each round, deals two cards to the player and one upcard to the dealer, and resets the game state.

```python
def __init__(self, num_decks=1):
    self.num_decks = num_decks
    self.reset_deck()
    self.player_hand = []
    self.dealer_hand = []
    self.done = False

def reset_deck(self):
    suits = ['H', 'D', 'S', 'C']
    values = list(range(1, 14))  # 1=A, 11=J, 12=Q, 13=K
    self.deck = []
    for _ in range(self.num_decks):
        for suit in suits:
            for value in values:
                self.deck.append((value, suit))
    random.shuffle(self.deck)
```

Main Approach - Environment Code

- Game State Representation (get_obs)

  Outputs player state features at each step, including:

  Player total sum (player_sum)

  Soft hand status (is_soft)

  Dealer upcard (dealer_card)

  Number of player cards (num_cards)

```python
def get_obs(self):
    player_sum, is_soft = self.hand_value(self.player_hand)
    dealer_card = min(self.dealer_hand[0][0], 10)
    num_cards = len(self.player_hand)
    return {
        "player_sum": player_sum,
        "dealer_card": dealer_card,
        "is_soft": int(is_soft),
        "num_cards": num_cards
    }
```

```python
def player_hit(self):
    if self.done:
        return self.get_obs(), "game_over", True
    self.player_hand.append(self.draw_card())
    player_sum, _ = self.hand_value(self.player_hand)
    if player_sum > 21:
        self.done = True
        return self.get_obs(), "lose", True
    return self.get_obs(), None, False

def player_stand(self):
    return self.get_obs(), None, False

def dealer_draw_one(self):
    if self.done:
        return True
    self.dealer_hand.append(self.draw_card())
    return self.dealer_is_done()

def dealer_is_done(self):
    dealer_sum, dealer_soft = self.hand_value(self.dealer_hand)
    if dealer_sum >= 17 and not (dealer_sum == 17 and dealer_soft):
        self.done = True
        return True
    return False
```

- Player Action Handling

  Players can choose "hit" or "stand"; the system updates the game state accordingly:

  If "hit", draw a card and update the state.

  If "stand", the dealer plays automatically following standard rules (stop at 17 or higher unless it’s a soft 17).

- Game Result Evaluation (get_game_result)

  Compares player and dealer final hands to determine the outcome: win / lose / draw.

Main Approach-Expectimax Agent

Objective:

Compare expected outcomes of 'Hit' and 'Stand' to decide optimal move.

Approach:

- 1st Loop (Stand):
  - Loop through all dealer possible values (2–11).
  - If dealer < 17, assume dealer draws to 17.
  - Compute difference:
    Agent Value - Dealer Value
  - Accumulate differences to get Stand EV.
- 2nd Loop (Hit):
  - Agent draws all possible cards (excluding dealer's card).
  - Add drawn card to agent's hand.
  - Re-run 1st loop logic for each new agent value.
  - Accumulate differences to get Hit EV.
- Final Decision:
  - Compare averages:
    Hit EV vs Stand EV
  - Choose the action with the higher expected value.

Main Approach – Random Forest Classifier (RFC) Agent

Training Process (train_rfc_agent.py):

- Data Source: blackjack_simulator.csv
- Features: player_sum, dealer_card, is_soft, num_cards
- Label: 0=Hit, 1=Stand

```python
df = pd.read_csv("blackjack_simulator.csv", nrows=500000)
X, y = [], []
for idx, row in tqdm(df.iterrows(), total=len(df), desc="Processing Data"):
    try:
        hand = ast.literal_eval(row['initial_hand'])
        actions = ast.literal_eval(row['actions_taken'])[0]
        for action in actions:
            player_sum = sum(hand)
            is_soft = int(11 in hand and player_sum <= 21)
            features = [player_sum, int(row['dealer_up']), is_soft, len(hand)]
            label = 0 if action == 'H' else 1
            X.append(features)
            y.append(label)
            if action == 'H':
                hand.append(random.choice([2,3,4,5,6,7,8,9,10,11]))
    except:
        continue
```

Main Approach – Random Forest Classifier (RFC) Agent

Training Process (train_rfc_agent.py):
Random Forest Training Logic

- Train multiple decision trees (default: 5 trees, max depth 5) using Bagging for diversity.
- Each tree is trained on a bootstrap sample (random subset with replacement). Decision trees are built recursively:
  1. Use Gini impurity to find the best split.
  2. Create left/right branches until max depth or pure leaf.

```python
class DecisionTree:
    def _build_tree(self, X, y, depth):
        if depth >= self.max_depth or len(set(y)) == 1:
            return int(Counter(y).most_common(1)[0][0])

        feature, threshold = self._best_split(X, y)
        if feature is None:
            return int(Counter(y).most_common(1)[0][0])

        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask

        return {
            'feature': int(feature),
            'threshold': float(threshold),
            'left': self._build_tree(X[left_mask], y[left_mask], depth+1),
            'right': self._build_tree(X[right_mask], y[right_mask], depth+1)
        }

    def predict_one(self, x, node=None):
        if node is None:
            node = self.tree
        if not isinstance(node, dict):
            return node
        if x[node['feature']] <= node['threshold']:
            return self.predict_one(x, node['left'])
        else:
            return self.predict_one(x, node['right'])

    def predict(self, X):
        return np.array([self.predict_one(x) for x in X])
```

```python
class RandomForest:
    def __init__(self, n_trees=5, max_depth=5):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _ in tqdm(range(self.n_trees), desc="Training Trees"):
            indices = np.random.choice(len(X), len(X), replace=True)
            tree = DecisionTree(self.max_depth)
            tree.fit(X[indices], y[indices])
            self.trees.append(tree)

    def predict(self, X):
        all_preds = np.array([tree.predict(X) for tree in self.trees])
        final_preds = []
        for i in range(X.shape[0]):
            votes = all_preds[:, i]
            final_preds.append(int(Counter(votes).most_common(1)[0][0]))
        return np.array(final_preds)
```

```python
class DecisionTree:
    def __init__(self, max_depth=5):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        self.tree = self._build_tree(X, y, depth=0)

    def _gini(self, y):
        counts = Counter(y)
        impurity = 1.0
        for label in counts:
            prob = counts[label] / len(y)
            impurity -= prob ** 2
        return impurity

    def _best_split(self, X, y):
        best_gain = -1
        best_feature, best_threshold = None, None
        current_gini = self._gini(y)

        for feature in range(X.shape[1]):
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                left_mask = X[:, feature] <= threshold
                right_mask = ~left_mask
                if sum(left_mask) == 0 or sum(right_mask) == 0:
                    continue
                left_y, right_y = y[left_mask], y[right_mask]
                gain = current_gini - (
                    len(left_y) / len(y) * self._gini(left_y) +
                    len(right_y) / len(y) * self._gini(right_y)
                )
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold
        return best_feature, best_threshold
```

Main Approach – Random Forest Classifier (RFC) Agent

Training Process (train_rfc_agent.py)

- Output: Model saved as JSON (rfc_model.json) with feature, threshold, left, right structure

Inference Logic (rfc_agent.py)

- Loads rfc_model.json
- For each state, traverse trees and collect votes
- Final action by majority voting: Hit/Stand

```
X = np.array(X)
y = np.array(y)

model = RandomForest(n_trees=5, max_depth=5)
model.fit(X, y)

forest_structure = [tree.tree for tree in model.trees]
forest_structure = convert_to_builtin(forest_structure)

with open('rfc_model.json', 'w') as f:
    json.dump(forest_structure, f)

print("RFC model trained and saved as rfc_model.json")
```

```
import json

class RFCAgent:
    def __init__(self, model_file):
        with open(model_file, 'r') as f:
            self.forest = json.load(f)

    def act(self, player_sum, dealer_card, is_soft, num_cards):
        features = [player_sum, dealer_card, int(is_soft), num_cards]
        votes = []
        for tree in self.forest:
            votes.append(self.predict_tree(features, tree))
        decision = max(set(votes), key=votes.count)
        return "hit" if decision == 0 else "stand"

    def predict_tree(self, x, node):
        if not isinstance(node, dict):
            return node
        if x[node['feature']] <= node['threshold']:
            return self.predict_tree(x, node['left'])
        else:
            return self.predict_tree(x, node['right'])
```

Main Approach – NEAT Agent

- NEAT (NeuroEvolution of Augmenting Topologies) evolves both network topology and weights for the Blackjack AI.
- Each genome = a neural network:
  Node genes (key, bias, layer: input/output/hidden)
  Connection genes (src, dst, weight, enabled, innovation)
- Mutation operations: Change weights, change biases, add connections, add nodes
- Crossover: Combine two parents based on innovation numbers
- Speciation: Group genomes by similarity to preserve diversity
- Fitness: Total reward from 1000 Blackjack games (input features: player sum, dealer card, is soft, num cards)
- Output: 1 node → "hit" (if >0.5) or "stand“
- Final model: Saved as JSON (node biases, layers, connection weights) for inference.

Main Approach – NEAT Agent

Node / Connection Structures

```python
class Node:
    def __init__(self, key, bias=None, layer=None):
        self.key = key
        self.bias = random.uniform(-1, 1) if bias is None else bias
        self.layer = layer

class Conn:
    def __init__(self, src, dst, weight=None, enabled=True, tag=None):
        self.src = src
        self.dst = dst
        self.weight = random.uniform(-1, 1) if weight is None else weight
        self.enabled = enabled
        self.tag = tag
```

Crossover

```python
@staticmethod
def crossover(p1, p2, cfg):
    if p2.fitness > p1.fitness: p1, p2 = p2, p1
    child = Genome(None, cfg)
    child.nodes = {k: Node(k, n.bias, n.layer) for k, n in p1.nodes.items()}
    child.conns = {}
    for k, c in p1.conns.items():
        other = p2.conns.get(k, c)
        chosen = other if random.random() < 0.5 else c
        child.conns[k] = Conn(chosen.src, chosen.dst, chosen.weight, chosen.enabled, chosen.tag)
    return child
```

Mutation Logic

```python
def mutate(self):
    if random.random() < 0.8: self.mutate_weights()
    if random.random() < 0.8: self.mutate_biases()
    if random.random() < 0.03: self.add_conn()
    if random.random() < 0.05: self.add_node()
```

Evaluate

```python
def evaluate(self):
    env = BlackjackEnv()
    for g in self.genomes.values():
        net = Network(g)
        total = 0
        for _ in range(1000):
            state, done = env.reset(), False
            while not done:
                x = [state["player_sum"]/21, state["dealer_card"]/10, int(state["is_soft"]), len(env.player_hand)/5]
                action = "hit" if net.activate(x)[0] > 0.5 else "stand"
                state, reward, done = env.step(action)
                total += reward
        g.fitness = total
```

Inference Logic

```python
class NEATAgent:
    def __init__(self, model_path="neat_model.json"):
        with open(model_path, encoding="utf-8") as f: data = json.load(f)
        self.net = Network(data)
    def act(self, player_sum, dealer_card, is_soft, num_cards):
        x = [player_sum/21, dealer_card/10, int(is_soft), num_cards/5]
        return "hit" if self.net.activate(x) > 0.5 else "stand"
```

Main Approach-Graphical User Interface (GUI) (blackjack_gui.py)

GUI Features

Component	Description
Mode Selection	Dropdown menu: Manual, Random, Basic, RFC, NEAT
Manual Play Buttons	Hit, Stand
AI Simulation Control	Auto-run games in batch, update results without dialog boxes
Card Display	Visual display of cards using **Vector Playing Cards** images
Game Status Display	Shows player/dealer hands, scores, wins, losses, draws
Result Dialog (Manual)	After each manual game, popup shows result and allows new game

Integration Logic:

GUI interacts with BlackjackEnv to get game state and render visuals.

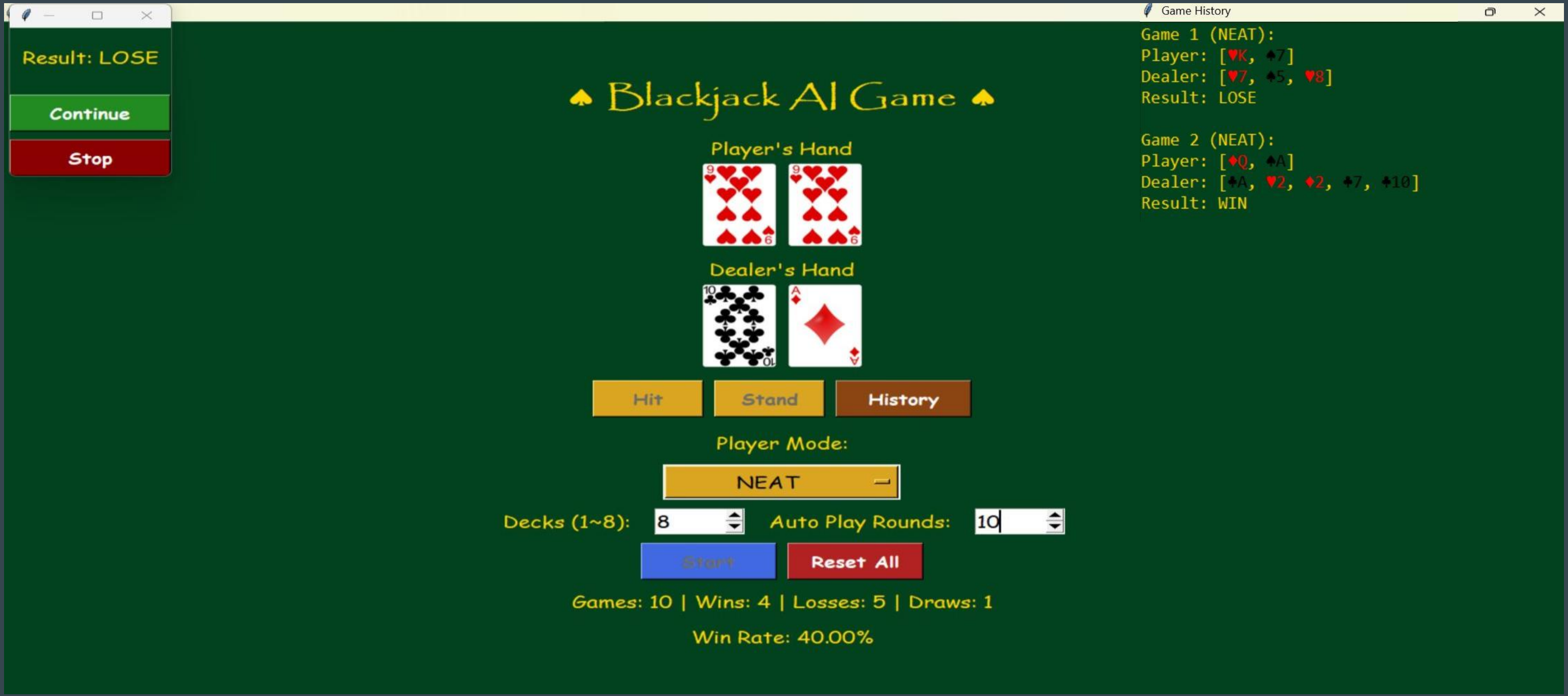
When AI mode is selected, the GUI runs simulations automatically.

The GUI updating statistics and history in real-time.

Manual mode supports full gameplay with user decisions.

Each agent is tested over 100 trials, each trial simulating 1000 games.
Metrics collected: Win rate. Results visualized as line plots in compare.png

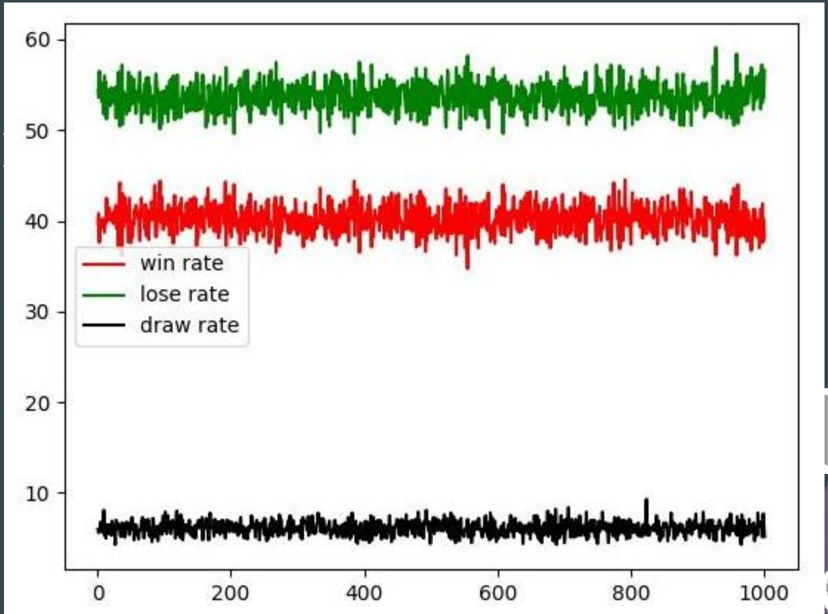
Main Approach-Graphical User Interface (GUI) (blackjack_gui.py)



Expectimax Agent

Win Rate:

- Approximately 40% win rate
- Notable, considering Blackjack's high uncertainty and the agent's lack of advanced features (e.g., card counting)



Stability:

- Variance < 1.5% across multiple runs
- Indicates a consistently performing agent, not overly affected by random fluctuations

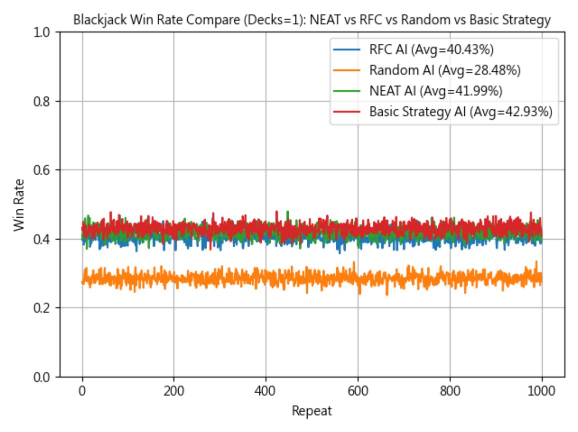
Interpretation:

- Expectimax performs well despite the game's probabilistic nature
- Suggests strong decision-making logic in terms of evaluating *Hit* vs *Stand*

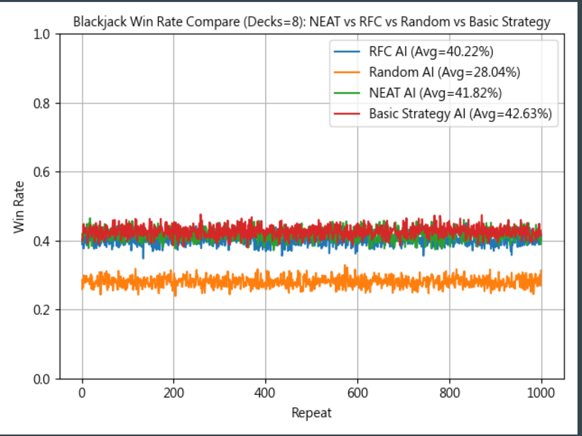
Result and Analysis

AI Agent

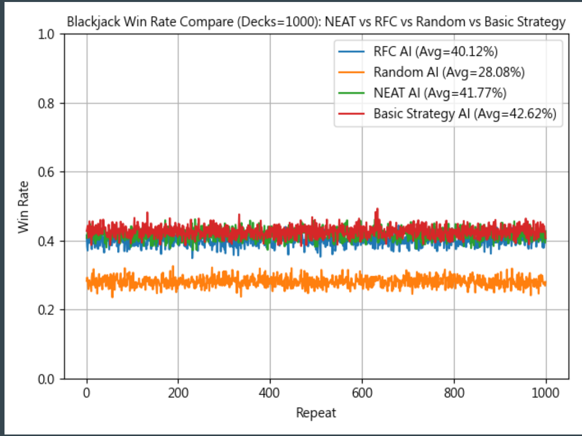
Deck=1



Deck=8



Deck=1000



The Better Model?

After testing, the result is:

When comparing winning rate:

Basic strategy > NEAT > Expectimax > RFC > Random

Thank You!