**פרויקט משחק מבוך דינמי עם למידת מכונה ואלגוריתם A\***

דנה שלמוביץ' 316110642

אליעד הופמן 209014018

ליאור אקרמן 302346895

חי אליאסי 204726913

**לינקים:**

<https://youtu.be/sym-FfJMSY0> - מצגת

<https://youtu.be/cyhHpyDKKGM> - הצגת המשחק

סרט של המשחק. )גם להגיש לאתר הקורס וגם לעלות ל-YouTube כ-Public). יש לצרף שני הלינקים למסמך הסופי.

**סיכום כללי:**

#### **תיאור הבעיה**

המטרה הייתה ליצור משחק מבוך דינמי שבו שני שחקנים מנסים להגיע למטרה שלהם תוך כדי שהם יכולים להניח מחסומים (logs) במבוך כדי להפריע לשחקן היריב. הפרויקט משלב למידת מכונה ואלגוריתם חיפוש כדי להחליט על הפעולות הבאות של השחקנים.

#### **פתרון הבעיה**

במשחק זה כל שחקן מנסה להגיע למיקום המטרה שלו, תוך כדי שהוא יכול להניח מחסומים (logs) שיקשו על היריב להגיע למטרה שלו. השחקנים משתמשים באלגוריתם A\* כדי למצוא את הדרך הקצרה ביותר למטרה שלהם, ולמידת מכונה כדי להחליט מתי ואיפה להניח את המחסומים.

#### **אלגוריתמים בשימוש**

1. **אלגוריתם A**\*:
   * אלגוריתם חיפוש זה משמש למציאת המסלול הקצר ביותר בין מיקום השחקן למיקום המטרה שלו.
   * האלגוריתם משתמש בפונקציית הערכה שמשלבת את המרחק המצטבר מהתחלה למיקום הנוכחי ואת המרחק הישר למטרה.
2. **למידת מכונה (MLPRegressor)**:
   * כל שחקן משתמש במודל MLP (Multi-Layer Perceptron) כדי ללמוד ולהחליט על הפעולה הבאה שלו, בין אם להניח מחסום ובין אם לנוע לעבר המטרה.
   * המודל מתאמן על נתוני משחקים קודמים כדי לשפר את היכולת שלו להחליט החלטות נכונות.
   * במהלך המשחק, השחקן שומר את הנתונים מהמשחק הנוכחי ומאמן את המודל מחדש על הנתונים המצטברים.

#### **תהליך העבודה במשחק**

1. **התחלת המשחק**:
   * המשחק מתחיל בבחירת תמונות לשחקנים.
   * לאחר מכן, השחקנים ממוקמים בנקודות ההתחלה שלהם (אחד בפינה העליונה השמאלית והשני בפינה התחתונה הימנית).
2. **מהלך המשחק**:
   * בכל תור, שחקן יכול לבחור להניח מחסום או לנוע לעבר המטרה שלו.
   * אם השחקן מחליט להניח מחסום, הוא מנסה למצוא את המקום שבו הנחת המחסום תהיה הכי מועילה (כלומר, תפריע ליריב להגיע למטרה שלו).
   * אם השחקן מחליט לנוע, הוא משתמש באלגוריתם A\* כדי למצוא את הדרך הקצרה ביותר למטרה שלו ונע לאורך המסלול שנמצא.
3. **סיום המשחק**:
   * המשחק נגמר כאשר אחד השחקנים מגיע למטרה שלו, או כאשר מספר תורות מקסימלי נגיע.
   * המידע מהמשחק נשמר בקובץ JSON לצורך אימון עתידי של המודל.

#### **שמירת וטעינת נתונים**

* במהלך כל משחק, המידע נאסף ונשמר בקובץ JSON לצורך אימון המודל.
* לפני תחילת משחק חדש, הנתונים מקובץ ה-JSON נטענים והמודל מתאמן מחדש על כל הנתונים המצטברים.

#### **אתגרים ופתרונות**

* **קביעת מיקום הנחת המחסומים**:
  + נעשה שימוש במנגנון אקראי כדי לקבוע מיקומים פוטנציאליים למחסומים ולאחר מכן נבדקה השפעת כל מיקום על המסלול של השחקן היריב.
* **מניעת תורות אין סופיים**:
  + הוגדר מספר מקסימלי של תורות למשחק כדי למנוע משחקים אינסופיים.
* **שיפור המודל**:
  + המודל מתאמן מחדש לאחר כל משחק על כל הנתונים המצטברים כדי לשפר את הדיוק שלו בהחלטות עתידיות.

**הקוד של המשחק:**

import pygame

import sys

import json

from queue import PriorityQueue

import random

import numpy as np

from sklearn.neural\_network import MLPRegressor

# Initialize Pygame

pygame.init()

# Constants

WIDTH, HEIGHT = pygame.display.Info().current\_w, pygame.display.Info().current\_h

GRID\_SIZE = 10

CELL\_SIZE = min(WIDTH // GRID\_SIZE, HEIGHT // GRID\_SIZE) # This ensures the grid fits within the window

# Colors

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

BLUE = (0, 0, 255)

BROWN = (139, 69, 19)

GREEN = (0, 255, 0)

YELLOW = (255, 255, 0)

GREY = (169, 169, 169)

DARK\_GREY = (105, 105, 105)

BACKGROUND\_COLOR = (200, 225, 255)

BUTTON\_COLOR = (70, 130, 180)

BUTTON\_HOVER\_COLOR = (100, 149, 237)

TEXT\_COLOR = (255, 255, 255)

ARROW\_COLOR = (50, 50, 50)

# Set up the display

screen = pygame.display.set\_mode((WIDTH, HEIGHT), pygame.FULLSCREEN)

pygame.display.set\_caption("Maze Game")

font = pygame.font.Font(pygame.font.get\_default\_font(), 24)

menu\_font = pygame.font.Font(pygame.font.get\_default\_font(), 48)

button\_font = pygame.font.Font(pygame.font.get\_default\_font(), 36)

arrow\_font = pygame.font.Font(pygame.font.get\_default\_font(), 48)

# Load the door and player images

door\_image = pygame.image.load('door.png')

player1\_images = [

pygame.image.load('animalplayer1.jpg'),

pygame.image.load('animalplayer2.jpg'),

pygame.image.load('animalplayer3.jpg'),

pygame.image.load('animalplayer4.jpg')

]

player2\_images = [

pygame.image.load('animalplayer1.jpg'),

pygame.image.load('animalplayer2.jpg'),

pygame.image.load('animalplayer3.jpg'),

pygame.image.load('animalplayer4.jpg')

]

# Scale the images to fit the grid cells

door\_image = pygame.transform.scale(door\_image, (CELL\_SIZE, CELL\_SIZE))

player1\_images = [pygame.transform.scale(img, (CELL\_SIZE, CELL\_SIZE)) for img in player1\_images]

player2\_images = [pygame.transform.scale(img, (CELL\_SIZE, CELL\_SIZE)) for img in player2\_images]

# Initialize the chosen player images

player1\_img = player1\_images[0]

player2\_img = player2\_images[1]

class Player:

def \_\_init\_\_(self, x, y, image, goal):

self.x = x # X coordinate

self.y = y # Y coordinate

self.image = image # Player image

self.logs = 5 # Number of logs player can place

self.goal = goal # Goal position

self.path = [] # Path to follow

self.model = MLPRegressor(hidden\_layer\_sizes=(50, 50), max\_iter=1000, random\_state=42) # Machine learning model

self.train\_data = [] # Training data for the model

self.train\_model() # Train the model with initial data

self.logs\_placed = 0 # Number of logs placed

self.moves\_left = 5 # Moves left in the current turn

self.has\_jumped = False # Flag to check if player has jumped

def move(self):

# Move player to the next position in the path

if self.path is not None and len(self.path) > 1 and self.moves\_left > 0:

next\_x, next\_y = self.path[1]

if abs(next\_x - self.x) + abs(next\_y - self.y) > 1:

self.has\_jumped = True

self.x, self.y = next\_x, next\_y

self.path = self.path[1:]

self.moves\_left -= 1

def draw(self):

# Draw the player on the screen

screen.blit(self.image, (self.x \* CELL\_SIZE, self.y \* CELL\_SIZE))

def train\_model(self):

# Train the machine learning model

if len(self.train\_data) > 100:

X, y = zip(\*self.train\_data)

self.model.fit(X, y)

else:

X = np.random.rand(1000, 7)

y = np.random.rand(1000)

self.model.fit(X, y)

def load\_training\_data(self, data):

# Load training data for the model

self.train\_data.extend(data)

self.train\_model()

def decide\_action(self, opponent, logs):

# Decide the next action for the player

my\_distance = abs(self.x - self.goal[0]) + abs(self.y - self.goal[1])

opponent\_distance = abs(opponent.x - opponent.goal[0]) + abs(opponent.y - opponent.goal[1])

X = np.array([[self.x, self.y, opponent.x, opponent.y, self.logs, my\_distance, opponent\_distance]])

prediction = self.model.predict(X)[0]

if prediction > 0.7 and self.logs > 0:

action = 'place\_log'

elif prediction > 0.3:

action = 'hop'

else:

action = 'move'

self.train\_data.append((X[0], 1 if action == 'place\_log' else (0.5 if action == 'hop' else 0)))

return action

class Log:

def \_\_init\_\_(self, x, y, horizontal):

self.x = x # X coordinate

self.y = y # Y coordinate

self.horizontal = horizontal # Orientation of the log

def draw(self):

# Draw the log on the screen

if self.horizontal:

pygame.draw.rect(screen, BROWN,

(self.x \* CELL\_SIZE, self.y \* CELL\_SIZE + CELL\_SIZE // 4, CELL\_SIZE \* 2, CELL\_SIZE // 2))

else:

pygame.draw.rect(screen, BROWN,

(self.x \* CELL\_SIZE + CELL\_SIZE // 4, self.y \* CELL\_SIZE, CELL\_SIZE // 2, CELL\_SIZE \* 2))

def initialize\_players():

# Initialize players and logs

global player1, player2, logs

player1 = Player(0, 0, player1\_img, (GRID\_SIZE - 1, GRID\_SIZE - 1))

player2 = Player(GRID\_SIZE - 1, GRID\_SIZE - 1, player2\_img, (0, 0))

logs = []

player1.logs\_placed = 0

player2.logs\_placed = 0

player1.moves\_left = 5

player2.moves\_left = 5

player1.has\_jumped = False

player2.has\_jumped = False

def draw\_grid():

# Draw the grid on the screen

for x in range(0, GRID\_SIZE \* CELL\_SIZE, CELL\_SIZE):

pygame.draw.line(screen, DARK\_GREY, (x, 0), (x, GRID\_SIZE \* CELL\_SIZE))

for y in range(0, GRID\_SIZE \* CELL\_SIZE, CELL\_SIZE):

pygame.draw.line(screen, DARK\_GREY, (0, y), (GRID\_SIZE \* CELL\_SIZE, y))

def draw(mode):

# Draw the game elements on the screen

screen.fill(BACKGROUND\_COLOR)

draw\_grid()

for log in logs:

log.draw()

screen.blit(door\_image, (0 \* CELL\_SIZE, 0 \* CELL\_SIZE))

screen.blit(door\_image, ((GRID\_SIZE - 1) \* CELL\_SIZE, (GRID\_SIZE - 1) \* CELL\_SIZE))

player1.draw()

player2.draw()

for player in [player1, player2]:

if player.path:

for i in range(len(player.path) - 1):

start = player.path[i]

end = player.path[i + 1]

start\_pos = (start[0] \* CELL\_SIZE + CELL\_SIZE // 2, start[1] \* CELL\_SIZE + CELL\_SIZE // 2)

end\_pos = (end[0] \* CELL\_SIZE + CELL\_SIZE // 2, end[1] \* CELL\_SIZE + CELL\_SIZE // 2)

pygame.draw.line(screen, GREEN if player == player1 else YELLOW, start\_pos, end\_pos, 2)

info\_text = f"Player 1 Logs: {player1.logs}\nPlayer 2 Logs: {player2.logs}"

info\_surface = font.render(info\_text, True, BLACK)

screen.blit(info\_surface, (GRID\_SIZE \* CELL\_SIZE + 20, 20))

phase\_text = f"Current Phase: {'Log Placement' if mode == 'place\_log' else 'Movement'}"

phase\_surface = font.render(phase\_text, True, BLACK)

screen.blit(phase\_surface, (GRID\_SIZE \* CELL\_SIZE + 20, 80))

if mode == 'move':

moves\_text = f"Moves Left - Player 1: {player1.moves\_left}, Player 2: {player2.moves\_left}"

moves\_surface = font.render(moves\_text, True, BLACK)

screen.blit(moves\_surface, (GRID\_SIZE \* CELL\_SIZE + 20, 120))

pygame.display.flip()

def heuristic(a, b):

# Heuristic function for A\* algorithm

return abs(b[0] - a[0]) + abs(b[1] - a[1])

def get\_neighbors(x, y, current\_logs, can\_hop, has\_jumped):

# Get the neighboring cells for A\* algorithm

neighbors = []

for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < GRID\_SIZE and 0 <= new\_y < GRID\_SIZE:

if not any(log.x <= new\_x < log.x + (2 if log.horizontal else 1) and

log.y <= new\_y < log.y + (2 if not log.horizontal else 1) for log in current\_logs):

neighbors.append((new\_x, new\_y))

elif can\_hop and not has\_jumped:

hop\_x, hop\_y = new\_x + dx, new\_y + dy

if 0 <= hop\_x < GRID\_SIZE and 0 <= hop\_y < GRID\_SIZE:

if not any(log.x <= hop\_x < log.x + (2 if log.horizontal else 1) and

log.y <= hop\_y < log.y + (2 if not log.horizontal else 1) for log in current\_logs):

neighbors.append((hop\_x, hop\_y))

return neighbors

def a\_star(start, goal, current\_logs, can\_hop, has\_jumped):

# A\* pathfinding algorithm

frontier = PriorityQueue()

frontier.put((0, start))

came\_from = {start: None}

cost\_so\_far = {start: 0}

while not frontier.empty():

current = frontier.get()[1]

if current == goal:

path = []

while current != start:

path.append(current)

current = came\_from[current]

path.append(start)

return list(reversed(path))

for next in get\_neighbors(\*current, current\_logs, can\_hop, has\_jumped):

new\_cost = cost\_so\_far[current] + 1

if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

cost\_so\_far[next] = new\_cost

priority = new\_cost + heuristic(goal, next)

frontier.put((priority, next))

came\_from[next] = current

return None

def logs\_overlap(new\_log, existing\_logs):

# Check if a new log overlaps with existing logs

for log in existing\_logs:

if new\_log.horizontal == log.horizontal:

if new\_log.horizontal:

if new\_log.y == log.y and (new\_log.x < log.x + 2 and log.x < new\_log.x + 2):

return True

else:

if new\_log.x == log.x and (new\_log.y < log.y + 2 and log.y < new\_log.y + 2):

return True

else:

if new\_log.horizontal:

if (log.x <= new\_log.x < log.x + 1 or log.x <= new\_log.x + 1 < log.x + 1) and \

(new\_log.y <= log.y < new\_log.y + 1 or new\_log.y <= log.y + 1 < new\_log.y + 1):

return True

else:

if (new\_log.x <= log.x < new\_log.x + 1 or new\_log.x <= log.x + 1 < new\_log.x + 1) and \

(log.y <= new\_log.y < log.y + 1 or log.y <= new\_log.y + 1 < new\_log.y + 1):

return True

return False

def place\_log(player, opponent):

# Place a log on the grid

if player.logs > 0:

for \_ in range(50):

horizontal = random.choice([True, False])

if horizontal:

x = random.randint(0, GRID\_SIZE - 2)

y = random.randint(0, GRID\_SIZE - 1)

else:

x = random.randint(0, GRID\_SIZE - 1)

y = random.randint(0, GRID\_SIZE - 2)

new\_log = Log(x, y, horizontal)

if not logs\_overlap(new\_log, logs) and \

not (x == player.x and y == player.y) and \

not (x == opponent.x and y == opponent.y) and \

not (x == 0 and y == 0) and \

not (x == GRID\_SIZE - 1 and y == GRID\_SIZE - 1):

temp\_logs = logs + [new\_log]

path1 = a\_star((player.x, player.y), player.goal, temp\_logs, True, False)

path2 = a\_star((opponent.x, opponent.y), opponent.goal, temp\_logs, True, False)

if path1 and path2:

logs.append(new\_log)

player.logs -= 1

return True

return False

def player\_turn(player, opponent, mode):

# Execute the player's turn

if mode == 'place\_log':

if player.logs\_placed < 5:

if place\_log(player, opponent):

player.logs\_placed += 1

elif mode == 'move':

if player.moves\_left > 0:

player.path = a\_star((player.x, player.y), player.goal, logs, True, player.has\_jumped)

if player.path: # Check if path exists

player.move()

else:

# If no path found, the player might be stuck

player.has\_jumped = False # Reset the jump flag

def check\_win\_condition(player):

# Check if the player has reached the goal

return (player.x, player.y) == player.goal

def clear\_logs():

# Clear the logs from the grid

global logs

logs = []

def play\_game():

# Main game loop

turn = 0

mode = 'place\_log'

running = True

winner = None

max\_turns = 1000

while running and turn < max\_turns:

for event in pygame.event.get():

if event.type == pygame.QUIT:

return None

if mode == 'place\_log' and player1.logs\_placed == 5 and player2.logs\_placed == 5:

mode = 'move'

player1.path = a\_star((player1.x, player1.y), player1.goal, logs, True, player1.has\_jumped)

player2.path = a\_star((player2.x, player2.y), player2.goal, logs, True, player2.has\_jumped)

player1.moves\_left = 5

player2.moves\_left = 5

player1.has\_jumped = False

player2.has\_jumped = False

if turn % 2 == 0:

player\_turn(player1, player2, mode)

if mode == 'move' and check\_win\_condition(player1):

winner = "Player 1"

running = False

else:

player\_turn(player2, player1, mode)

if mode == 'move' and check\_win\_condition(player2):

winner = "Player 2"

running = False

if mode == 'move' and (player1.moves\_left == 0 and player2.moves\_left == 0):

mode = 'place\_log'

clear\_logs()

player1.logs = 5

player2.logs = 5

player1.logs\_placed = 0

player2.logs\_placed = 0

turn += 1

draw(mode)

pygame.time.Clock().tick(2)

if winner:

print(f"{winner} wins!")

elif turn >= max\_turns:

print("Game ended in a draw (max turns reached)")

else:

print("Game ended without a winner")

save\_game\_data(player1, player2, logs, winner)

return winner

def save\_game\_data(player1, player2, logs, winner):

# Save game data to a JSON file

game\_data = {

'player1': {'x': player1.x, 'y': player1.y, 'logs': player1.logs},

'player2': {'x': player2.x, 'y': player2.y, 'logs': player2.logs},

'logs': [{'x': log.x, 'y': log.y, 'horizontal': log.horizontal} for log in logs],

'winner': winner,

'training\_data': {

'player1': player1.train\_data,

'player2': player2.train\_data

}

}

try:

with open('game\_data.json', 'r') as file:

data = json.load(file)

except (FileNotFoundError, json.JSONDecodeError):

data = []

data.append(game\_data)

with open('game\_data.json', 'w') as file:

json.dump(data, file, indent=4)

def load\_training\_data():

# Load training data from a JSON file

try:

with open('game\_data.json', 'r') as file:

data = json.load(file)

except (FileNotFoundError, json.JSONDecodeError):

data = []

player1\_data = []

player2\_data = []

for game in data:

player1\_data.extend(game['training\_data']['player1'])

player2\_data.extend(game['training\_data']['player2'])

return player1\_data, player2\_data

def draw\_button(text, rect, is\_hovered=False):

# Draw a button on the screen

color = BUTTON\_HOVER\_COLOR if is\_hovered else BUTTON\_COLOR

pygame.draw.rect(screen, color, rect, border\_radius=10)

pygame.draw.rect(screen, BLACK, rect, 2, border\_radius=10)

text\_surface = button\_font.render(text, True, TEXT\_COLOR)

text\_rect = text\_surface.get\_rect(center=rect.center)

screen.blit(text\_surface, text\_rect)

def draw\_text(text, position, font, color=BLACK):

# Draw text on the screen

text\_surface = font.render(text, True, color)

text\_rect = text\_surface.get\_rect(center=position)

screen.blit(text\_surface, text\_rect)

def draw\_arrows(num\_games\_rect):

# Draw arrows for adjusting the number of games

left\_arrow = arrow\_font.render("<", True, ARROW\_COLOR)

right\_arrow = arrow\_font.render(">", True, ARROW\_COLOR)

left\_rect = left\_arrow.get\_rect(midright=(num\_games\_rect.left - 10, num\_games\_rect.centery))

right\_rect = right\_arrow.get\_rect(midleft=(num\_games\_rect.right + 10, num\_games\_rect.centery))

screen.blit(left\_arrow, left\_rect)

screen.blit(right\_arrow, right\_rect)

return left\_rect, right\_rect

def choose\_player\_images():

# Allow the user to choose player images

global player1\_img, player2\_img

chosen1 = False

chosen2 = False

while not (chosen1 and chosen2):

screen.fill(BACKGROUND\_COLOR)

for i, img in enumerate(player1\_images):

screen.blit(img, (WIDTH // 4 - CELL\_SIZE // 2, HEIGHT // 4 + i \* (CELL\_SIZE + 10)))

for i, img in enumerate(player2\_images):

screen.blit(img, (WIDTH \* 3 // 4 - CELL\_SIZE // 2, HEIGHT // 4 + i \* (CELL\_SIZE + 10)))

draw\_text("Choose Player 1", (WIDTH // 4, HEIGHT // 4 - 40), menu\_font)

draw\_text("Choose Player 2", (WIDTH \* 3 // 4, HEIGHT // 4 - 40), menu\_font)

pygame.display.flip()

for event in pygame.event.get():

if event.type == pygame.QUIT:

pygame.quit()

sys.exit()

if event.type == pygame.MOUSEBUTTONDOWN:

for i, img in enumerate(player1\_images):

if WIDTH // 4 - CELL\_SIZE // 2 <= event.pos[0] <= WIDTH // 4 + CELL\_SIZE // 2:

if HEIGHT // 4 + i \* (CELL\_SIZE + 10) <= event.pos[1] <= HEIGHT // 4 + i \* (CELL\_SIZE + 10) + CELL\_SIZE:

player1\_img = img

chosen1 = True

for i, img in enumerate(player2\_images):

if WIDTH \* 3 // 4 - CELL\_SIZE // 2 <= event.pos[0] <= WIDTH \* 3 // 4 + CELL\_SIZE // 2:

if HEIGHT // 4 + i \* (CELL\_SIZE + 10) <= event.pos[1] <= HEIGHT // 4 + i \* (CELL\_SIZE + 10) + CELL\_SIZE:

player2\_img = img

chosen2 = True

def main\_menu():

# Display the main menu

start\_text = menu\_font.render("Maze Game", True, BLACK)

start\_rect = pygame.Rect((WIDTH // 2 - 150, HEIGHT // 2 - 100, 300, 50))

choose\_rect = pygame.Rect((WIDTH // 2 - 150, HEIGHT // 2 + 50, 300, 50))

exit\_rect = pygame.Rect((WIDTH // 2 - 150, HEIGHT // 2 + 150, 300, 50))

num\_games = 1

num\_games\_text = menu\_font.render(f"Number of Games: {num\_games}", True, BLACK)

num\_games\_rect = num\_games\_text.get\_rect(center=(WIDTH // 2, HEIGHT // 2))

while True:

screen.fill(BACKGROUND\_COLOR)

title\_rect = start\_text.get\_rect(center=(WIDTH // 2, HEIGHT // 2 - 200))

screen.blit(start\_text, title\_rect)

mouse\_pos = pygame.mouse.get\_pos()

is\_start\_hovered = start\_rect.collidepoint(mouse\_pos)

is\_choose\_hovered = choose\_rect.collidepoint(mouse\_pos)

is\_exit\_hovered = exit\_rect.collidepoint(mouse\_pos)

draw\_button("Start Game", start\_rect, is\_start\_hovered)

draw\_button("Choose Players", choose\_rect, is\_choose\_hovered)

draw\_button("Exit", exit\_rect, is\_exit\_hovered)

num\_games\_text = menu\_font.render(f"Number of Games: {num\_games}", True, BLACK)

num\_games\_rect = num\_games\_text.get\_rect(center=(WIDTH // 2, HEIGHT // 2))

screen.blit(num\_games\_text, num\_games\_rect)

left\_arrow\_rect, right\_arrow\_rect = draw\_arrows(num\_games\_rect)

pygame.display.flip()

for event in pygame.event.get():

if event.type == pygame.QUIT:

return 'exit', 0

if event.type == pygame.MOUSEBUTTONDOWN:

if start\_rect.collidepoint(event.pos):

return 'start', num\_games

elif choose\_rect.collidepoint(event.pos):

choose\_player\_images()

elif exit\_rect.collidepoint(event.pos):

return 'exit', 0

elif left\_arrow\_rect.collidepoint(event.pos):

num\_games = max(1, num\_games - 1)

elif right\_arrow\_rect.collidepoint(event.pos):

num\_games += 1

if event.type == pygame.KEYDOWN:

if event.key == pygame.K\_UP:

num\_games += 1

elif event.key == pygame.K\_DOWN:

num\_games = max(1, num\_games - 1)

num\_games\_text = menu\_font.render(f"Number of Games: {num\_games}", True, BLACK)

num\_games\_rect = num\_games\_text.get\_rect(center=(WIDTH // 2, HEIGHT // 2))

def main():

# Main function

player1\_data, player2\_data = load\_training\_data()

while True:

action, num\_games = main\_menu()

if action == 'exit':

pygame.quit()

sys.exit()

player1\_wins = 0

player2\_wins = 0

for game in range(num\_games):

print(f"Starting Game {game + 1}")

initialize\_players()

player1.load\_training\_data(player1\_data)

player2.load\_training\_data(player2\_data)

winner = play\_game()

if winner == "Player 1":

player1\_wins += 1

elif winner == "Player 2":

player2\_wins += 1

player1.train\_model()

player2.train\_model()

print(f"Final Score - Player 1: {player1\_wins}, Player 2: {player2\_wins}")

if \_\_name\_\_ == "\_\_main\_\_":

main()