فاطمه میرزائی کلانی ۴۰۰۲۴۳۰۷۵

تابع main_page صفحه اول که کاربر در آن بین minimax و alpha beta و reinforcement انتخاب می کند را نمایش می دهد. با فشار دادن دکمه ها تابع main فراخوانی میشود( که بازی با آن شروع می شود ).

```
def main_page(board, player):
    logo = pygame.image.load("logo1.png")
    button_image1 = pygame.image.load("alpha.png")
    button_image2 = pygame.image.load("minimax.png")
    button_image3 = pygame.image.load("rl.png")

    button_rect = logo.get_rect()
    button_rect1 = button_image1.get_rect()
    button_rect2 = button_image2.get_rect()
    button_rect3 = button_image3.get_rect()

    button_rect.centerx = WIN.get_width() // 2
    button_rect.bottom = WIN.get_height() - 500

    button_rect1.centerx = WIN.get_width() // 2
    button_rect1.bottom = WIN.get_height() - 400

    button_rect2.centerx = WIN.get_width() // 2
    button_rect2.bottom = WIN.get_height() - 300

    button_rect3.centerx = WIN.get_width() // 2
    button_rect3.bottom = WIN.get_height() - 200

    WIN.fill(WHITE)

    WIN.blit(logo, button_rect)
    WIN.blit(button_image1, button_rect1)
    WIN.blit(button_image2, button_rect2)
    WIN.blit(button_image3, button_rect3)

    pygame.display.update()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
            if event.type == pygame.MOUSEBUTTONDOWN and button_rect1.collidepoint(event.pos):
                main(board, player, "AlphaBeta")
            elif event.type == pygame.MOUSEBUTTONDOWN and button_rect2.collidepoint(event.pos):
                main(board, player, "MiniMax")
            elif event.type == pygame.MOUSEBUTTONDOWN and button_rect3.collidepoint(event.pos):
                main(board, player, "RL")
```

تابع main چک می کند نوبت کدام player است. بازی کننده 0 یعنی انسان و 1 یعنی AI. اگر player انسان باشد column ورودی را از آن می‌گیرد و اولین row خالی را برای board در نظر میگیرد. بعد چک می کند آیا player در حالت برنده است؟ اگر برنده بود متن You won را چاپ میکند. همچنین draw هم چک می کند.

اگر نوبت AI باشد اول چک می کند که نوع بازی چیست. سپس با توابع play_alphabeta یا play_minimax بازی میکند. و دوباره win و draw را چک میکند.

در تابع evaluate یک بار در هر سطر چهار تا چهار تا خانه‌ها را چک میکنیم و با تابع count_score امتیاز آن را حساب میکنیم.
برای هر ستون هم چهار تا چهار تا خانه ها را به همین روش چک میکنیم.
همچنین اگر مهره ای از AI در ستون 3 یعنی ستون وسط بود به امتیاز 6 تا اضافه میکنیم و اگر مهره human در آن بود از امتیاز 4 تا کم میکنیم. چون در ستون وسط از هر دو طرف امکان برد وجود دارد.

در تابع count_score اگر چهار تا مهره player کنار هم باشد 1000 امتیاز میدهیم. اما اگر چهار تا مهره opponent باشد 1000 امتیاز کم میکنیم. اگر سه تا مهره player باشد و یکی خالی باشد 30 امتیاز میگیرد. اگر دو تا مهره player و دو تا خالی باشد امتیاز 10 تا زیاد میشود. اگر رقیب همین حالات را داشته باشد از score کم میکنیم.

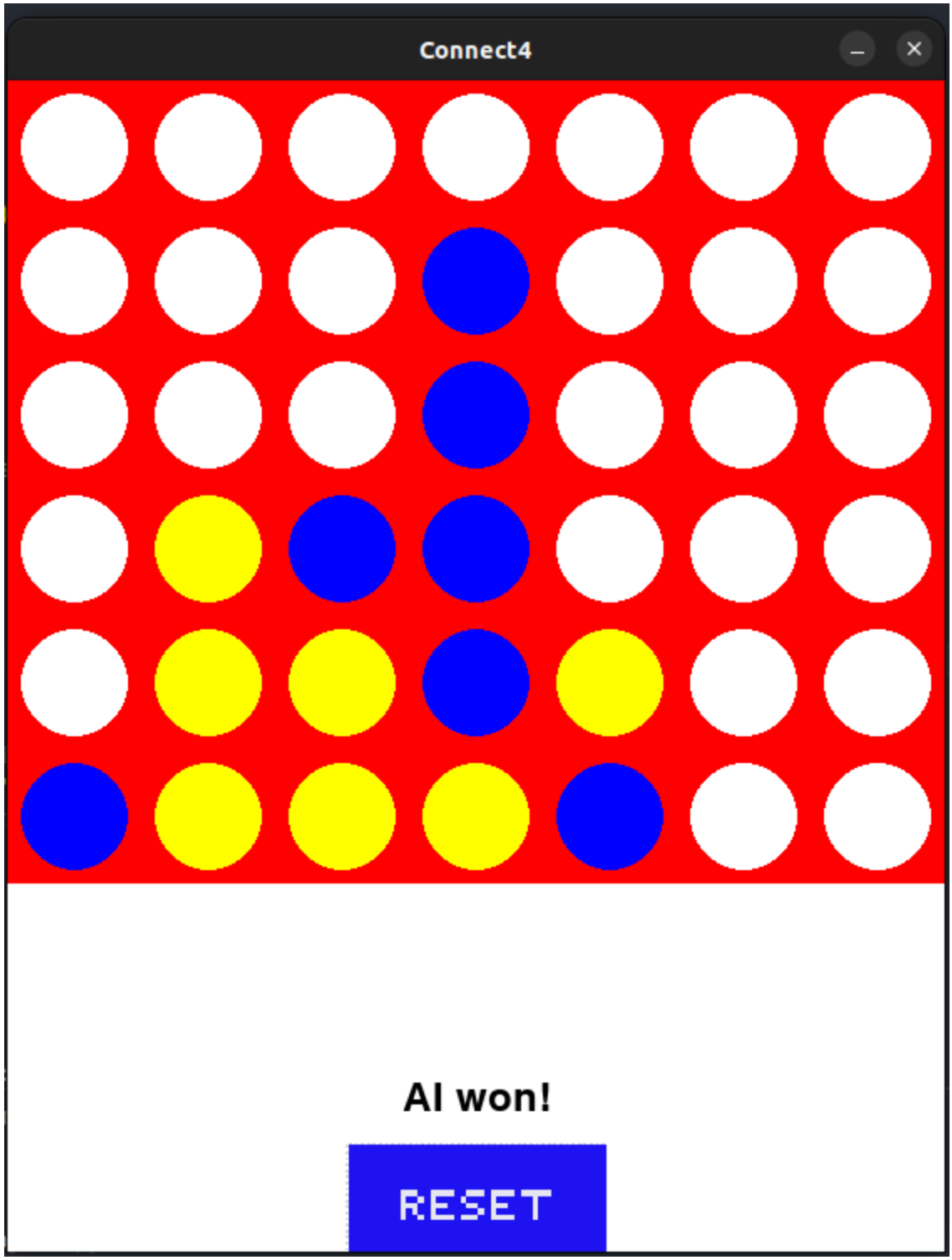
تابع check_draw چک میکند که آیا همه خانه ها پر شدند یا نه. اگر شده باشد draw است.
تابع check_win در هر سطر چهار تا چهار تا چک میکند که خانه ها پر شدند( از یک رنگ) یا نه. سپس خانه های هر ستون هم چهار تا چهار تا چک میکند.

تابع get_valid_columns شماره column هایی که خانه خالی دارند را در لیستی بر می گرداند.

تابع minimax در واقع به شکل بازگشتی با در نظر گرفتن درخت minimax، هیوریستیک هر گره را محاسبه میکند. در این تابع عمق را ۴ گرفتم.

تابع alpha beta مشابه همان minimax است فقط بر اساس دو متغیر هرس هم انجام میدهد. در این تابع عمق را مساوی ۶ گرفتم. زیرا کمتر از آن دقیق نیست و بیشتر از آن کند است

گرافیک بازی به این شکل است:


Connect4
AI won!
RESET

---

برای قسمت RL:

۱- یک agent را در فایل random_training ساختم و train کردم. در واقع ۱۰۰۰۰ بار agent با AI که به شکل رندوم حرکت انجام میدهد، بازی کرد. حالات این بازی خیلی زیاد است بخاطر همین فقط یک Q_table با سایز ۶*۷*۷ ساختم که state فعلی و حرکات ممکن به ازای هر state را نگه داری میکند. این q_table را بر اساس فرمول زیر آپدیت کردم:

```
def update_q_table(state, action, reward, next_state):
    learning_rate = 0.1
    discount_factor = 0.9

    q_table[state[0]][state[1]][action] += learning_rate * (reward + discount_factor * np.max(q_table[next_state[0]][next_state[1]]) - q_table[state[0]][state[1]][action])
```

همچنین یک متغیر epsilon هم در نظر گرفتم که در ابتدا زیاد است و رفته رفته کم میشود( وقتی زیاد باشد اجازه explore به شکل رندوم را با احتمال بیشتری میدهد).

اگر epsilon کمتر از مقدار رندوم بود، برای انتخاب action باید بزرگترین action در آن سطر و ستون در q_table را پیدا کنیم و انجام دهیم.

```
# epsilon-greedy strategy
def choose_action(state, epsilon):
    if random.uniform(0, 1) < epsilon:
        action = random.randint(0, NUM_ACTIONS - 1)
    else:
        row = state[0]
        col = state[1]
        action = np.argmax(q_table[row, col, :])
    return action
```

حال ۱۰۰۰۰ بار بازی را انجام میدهیم و همزمان با آن مقدار epsilon تغییر میکند:

```python
# Training loop
num_episodes = 10000
win = 0
lose = 0

for episode in range(num_episodes):
    reward = play_game(epsilon)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
    if reward == 1:
        win+=1
    elif reward == -1:
        lose+=1
print(f"while training: wining percent of trained agent:{win*100/num_episodes}")
print(f"while training: losing percent of trained agent:{lose*100/num_episodes}")
```

نتیجه زیر حاصل شد:

```
while training: wining percent of trained agent:78.36
while training: losing percent of trained agent:20.43
```

---

برای قسمت RL:

۲- یک agent را در فایل minimax_training ساختم که ۱۰۰۰۰ بار با یک agent (که بر اساس الگوریتم minimax عمل میکند) بازی کند.

قسمت های مربوط به آپدیت q_table و choose_action مانند بخش قبل است. تنها تفاوت در نحوه بازی حریف است. در اینجا حریف با minimax با عمق ۱ بازی میکند.

```python
# Training loop
num_episodes = 10000
win = 0
lose = 0
for episode in range(num_episodes):
    reward = play_game(epsilon)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
    if reward == 1:
        win+=1
    elif reward == -1:
        lose+=1
print(f"while training: wining percent of trained agent:{win*100/num_episodes}")
print(f"while training: losing percent of trained agent:{lose*100/num_episodes}")
```

نتیجه در هنگام train کردن:

```
while training: wining percent of trained agent:95.82
while training: losing percent of trained agent:4.18
```

همچنین برای تست کردن q_table بدست آمده، عامل train شده را در رقابت با حریف گذاشتم تا ببینم در چند درصد مواقع میبرد. حریف تا عمق یک minimax را در نظر میگیرد.

```python
# getting percentage of winning of trained model
trained_win = 0
AI_win = 0
max_index = np.unravel_index(np.argmax(q_table[5]), q_table[5].shape)
current_cell = [5, max_index[0]]
for _ in range(num_episodes):
    result = play_against(current_cell)
    if result == 1:
        trained_win+=1
    elif result == 2:
        AI_win += 1

print(f"trained agent winning rate:{trained_win * 100 / num_episodes}")
print(f"trained agent losing rate:{AI_win * 100 / num_episodes}")
```

نتیجه نهایی:

```
trained agent winning rate:100.0
trained agent losing rate:0.0
```

البته این مقدار در هر بار ران کردن تغییر میکند.

---

در نهایت میتوانیم بازی را در محیط گرافیکی با هر دو agent امتحان کنیم(برای قسمت RL):

```python
q_table = np.load('Q_table_random.npy')
# q_table = np.load('Q_table_minimax.npy')  # uncomment this if you wanna test it
```