



گرداورندگان: آراس ولیزاده - امیرسامان
رستم بیگی

دانشکده مهندسی و علوم کامپیوتر
گزارش فاز اول پروژه هوش مصنوعی و سیستم
های خبره - دکتر سلیمانی بدر
پاییز ۱۴۰۳

داخل این کد یه عامل هوشمند برای بازی "Hand of the King" طراحی کردیم که از الگوریتم مینیمکس با هرس آلفا-بتا استفاده می کنه تا بهترین حرکت رو برای بازیکن پیدا کنه.

عامل هوشمند چیه؟

عامل هوشمند یه برنامه یا کد کامپیوتريه که می تونه وضعیت بازی رو تحلیل کنه و بر اساس استراتژی های تعریف شده، بهترین حرکت ممکن رو انتخاب کنه. اینجا عامل ما:

- حرکت های ممکن رو بررسی می کنه.
- امتیاز وضعیت های مختلف بازی رو می سنجه.
- حرکتی رو انتخاب می کنه که بیشترین احتمال برد رو داشته باشه.

الگوریتم مینیمکس چیه؟

الگوریتم مینیمکس یکی از روش های استاندارد برای حل مسائل بازی های نوبتیه، مثل شطرنج یا این بازی توی این الگوریتم:

- بازیکن ما (عامل هوشمند) تلاش می کنه امتیاز رو ماکزیمم کنه.
- حریف تلاش می کنه امتیاز رو مینیمیمم کنه.
- مینیمکس این تعامل رو شبیه سازی می کنه و وضعیت هایی که هر بازیکن می تونه انتخاب کنه رو بررسی می کنه.

هرس آلفا-بتا چیه؟

هرس آلفا-بta یه بهبود برای الگوریتم مینیمکس که باعث می شه شاخه هایی که نتیجه هی قطعی دارن، بررسی نشن. این کار دو تا مزیت داره:

- سرعت بیشتر: شاخه های بی فایده حذف می شن.
- بهره وری بالاتر: فقط وضعیت هایی که اهمیت دارن بررسی می شن.

مثلاً اگه توی یه بازی وضعیت A امتیاز بهتری نسبت به وضعیت B داشته باشه، نیازی نیست بقیه حرکت های B رو بررسی کنیم.

چرا مینیمکس اینجا خوب کار می کنه؟

- بازی "Hand of the King" یه بازی نوبتیه که حرکت های ممکنش قابل پیش بینیه.
- عامل هوشمند می تونه حرکت های بعدی خودش و حریف رو شبیه سازی کنه.
- هدف اینه که هر حرکتی که انتخاب می کنه، بیشترین امتیاز رو در آینده براش بیاره.

حالا میتونیم بریم سراغ اینکه بخش های مختلف کد رو توضیح بدیم:

```
def find_varys(cards):
    ...
    Finds the location of Varys on the board.
Parameters:
    cards (list): list of Card objects
Returns:
    varys_location (int): location of Varys
...
varys = [card for card in cards if card.get_name() == 'Varys']
return varys[0].get_location()
```

پیدا کردن مکان واریس (تابع `find_varys`)

این تابع توی لیست کارت ها می گردد و کارت "Varys" رو پیدا می کنه. بعد مکانش رو (که یه عدده و نشون دهنده جای کارت توی برد) بر می گردونه. از این برای مشخص کردن حرکت های معتبر استفاده می کنیم.

```
● ● ●  
def get_valid_moves(cards):  
    ...  
    Gets the possible moves for the player.  
  
    Parameters:  
        cards (list): list of Card objects  
  
    Returns:  
        moves (list): list of possible move  
    ...  
    varys_location = find_varys(cards)  
    varys_row, varys_col = varys_location // 6, varys_location % 6  
    moves = []  
  
    for card in cards:  
        if card.get_name() == 'Varys':  
            continue  
  
        row, col = card.get_location() // 6, card.get_location() % 6  
        if row == varys_row or col == varys_col:  
            moves.append(card.get_location())  
  
    return moves
```

پیدا کردن حرکت‌های معتبر (تابع get_valid_moves)

این تابع مشخص می‌کنه واریس توکدوم جهت‌ها (راست، چپ، بالا یا پایین) می‌تونه حرکت کنه:

- مکان واریس رو می‌گیره.
- کارت‌هایی که توی همون ردیف یا ستون هستن رو پیدا می‌کنه.
- یه لیست از مکان‌های ممکن رو برمی‌گدونه.

```

1 def evaluate_state(cards, player1, player2):
2     """
3         Calculates a heuristic score for the current game state by evaluating the ba
4             nners
5             captured by both players and the number of cards they control.
6
7             Parameters:
8                 cards (list): A list of `Card` objects representing the current game sta
9                     te.
10                player1 (Player): An object representing Player 1, including banners and
11                    cards.
12                player2 (Player): An object representing Player 2, including banners and
13                    cards.
14
15                Returns:
16                    int: A score indicating the favorability of the game state for Player
17                        1. Positive scores favor Player 1; negative scores favor Player 2.
18
19                banner_weights = {
20                    'Stark': 9, 'Greyjoy': 7, 'Lannister': 7, 'Targaryen': 6, 'Baratheon': 5,
21                    'Tyrell': 9, 'Tully': 10
22                }
23                banner_cards = {
24                    'Stark': 8, 'Greyjoy': 7, 'Lannister': 6, 'Targaryen': 5, 'Baratheon': 4,
25                    'Tyrell': 3, 'Tully': 2
26                }
27
28                score = 0
29
30                #calculate score for player 1
31                for house in player1.get_banners():
32                    if player1.get_banners()[house]:
33                        score += banner_weights[house] * 10
34                    num_cards = 0
35                    for card in player1.get_cards():
36                        if card == house:
37                            num_cards += 1
38                    if num_cards <= banner_cards[house] // 2 + 1:
39                        score += num_cards * 2
40
41                # Calculate score for Player 2
42                for house in player2.get_banners():
43                    if player2.get_banners()[house]:
44                        score -= banner_weights[house] * 10
45                    num_cards = 0
46                    for card in player2.get_cards():
47                        if card == house:
48                            num_cards += 1
49                    if num_cards > banner_cards[house] // 2 + 1:
50                        penalty = -(banner_cards[house] // 2)
51                    else:
52                        penalty = num_cards
53                    score -= penalty * 2
54
55                player1_moves = len(get_player_moves(player1.get_cards(), cards))
56                player2_moves = len(get_player_moves(player2.get_cards(), cards))
57                score += (player1_moves - player2_moves) * 3 # Reward more move options
58
59
60            return score

```

ارزیابی وضعیت بازی (تابع evaluate_state)

این تابع واقعاً قلب تصمیم‌گیری ایجنت ماست و کل عملکردش به این وابسته‌ست که بازی رو چجوری ارزیابی کنه. هدفش اینه که به هر وضعیت بازی یه امتیاز بده که این امتیاز مشخص کنه بازی چقدر به نفع بازیکن اول (عامل هوشمند ما) هست.

چطوری کار می‌کنه؟

1. امتیاز بنرها:

وقتی یه بازیکن بتونه یه خاندان رو کامل کنه (یعنی کارت‌های لازم اون خاندان رو جمع کنه)، بنز اون خاندان رو می‌گیره. اینجا تابع به هر بنز یه امتیاز خاص می‌ده که بر اساس ارزش خاندانه. مثلاً خاندان Stark امتیاز بالاتری داره چون تعداد کارت‌های بیشتری داره، در حالی که Tully با دو کارت تکمیل می‌شه ولی امتیاز کمتری داره.

- بازیکن اول بنز بگیره؟ امتیاز مثبت می‌گیره.

- بازیکن دوم بنز بگیره؟ امتیاز منفی برای بازیکن اول ثبت می‌شه.

2. امتیاز کارت‌ها:

اینجا تابع نگاه می‌کنه که هر بازیکن چند تا کارت از یه خاندان خاص داره.

- اگر کارت‌های بازیکن اول به تعداد کافی برسه که بنز رو بگیره، یه امتیاز اضافی می‌گیره.

- اگر تعداد کارت‌هاش کمتر از نصف کارت‌های اون خاندان باشه، امتیاز کمتری می‌گیره، چون تو رقابت ممکنه عقب بمونه.

3. حرکت‌های ممکن:

تابع تعداد حرکت‌های ممکن هر بازیکن رو هم چک می‌کنه. اگه بازیکن اول حرکت‌های بیشتری داشته باشه، امتیاز مثبت می‌گیره. ایده اینه که بازیکنی که گزینه‌های بیشتری داره، احتمالاً کنترل بیشتری رو بازی داره.

چرا مهمه؟

این تابع باعث می‌شود که بازی برآش بهتره. وقتی الگوریتم Minimax می‌خواهد حرکت‌های مختلف را امتحان کند، از این تابع برای مقایسه وضعیت‌ها استفاده می‌کند. اگر این ارزیابی درست کار کند، ایجنت می‌توانه تصمیم‌های بهتری بگیرد.

یه نکته باحال :)

اینجا ما به بنرهای کامل شده وزن بیشتری دادیم چون هدف نهایی بازی همین تصاحب بنرهاست. اما تعداد کارت‌ها و تعداد حرکت‌ها هم تاثیر دارد که ایجنت تو مسیر درست حرکت کند و شانسی برای برد به حریف نداشته باشد.

به زیون ساده: این تابع مثل یه مترکار می‌کند که وضعیت بازی رو می‌سنجه و می‌گوید کدام بازیکن جلوتره و چقدر جلوئه.

```

def simulate_move(cards, player1, player2, move, player):
    """
    Simulates a player's move and returns the resulting game state.

    Parameters:
        cards (list): A list of `Card` objects representing the current game state.
        player1 (Player): The Player 1 object including banners and cards.
        player2 (Player): The Player 2 object including banners and cards.
        move (int): The index of the card to move.
        player (int): The player making the move (1 for Player 1, 2 for Player 2).

    Returns:
        tuple: A tuple containing
            - list: A deep copy of the updated `cards` after the move.
            - Player: A deep copy of the updated `player1` object.
            - Player: A deep copy of the updated `player2` object.
    """
    new_cards, new_player1, new_player2 = map(
        copy.deepcopy, [cards, player1, player2])
    current_player = new_player1 if player == 1 else new_player2
    selected_house = make_move(new_cards, move, current_player)
    set_banners(new_player1, new_player2, selected_house, player)
    return new_cards, new_player1, new_player2

```

شبیه‌سازی حرکت (تابع simulate_move)

خب این تابع یه جورایی مثل یه ماشین زمان می‌مونه! وقتی یه حرکت رو انجام بدیم، این تابع بازی رو جلو می‌بره و وضعیت جدید رو بهمون نشون می‌ده، بدون اینکه واقعاً وضعیت اصلی بازی رو تغییر بده. از اینجا به بعد، ایجنت می‌تونه این وضعیت شبیه‌سازی شده رو بررسی کنه و ببینه این حرکت خوبه یا نه.

- اول از همه، وضعیت فعلی بازی شامل کارت‌ها، بازیکن‌ها و هر چیزی که تو بازی هست رو کپی می‌کنه. این کپی باعث می‌شه که وضعیت اصلی بازی دست‌نخورده بمونه.
- بعدش، حرکت موردنظر رو انجام می‌ده. این یعنی مثلاً Varys رواز جایی به جای دیگه می‌بره و کارت مربوطه رو می‌گیره.
- توی این حرکت، ممکنه بازیکن یه خاندان رو کامل کنه. وقتی این اتفاق بیفته، پرچم (بنر) اون خاندان به بازیکن داده می‌شه. برای همین، تابع set_banners هم صدا زده می‌شه تا وضعیت پرچم‌ها رو آپدیت کنه.
- در نهایت، وضعیت جدید بازی شامل کارت‌ها و اطلاعات بازیکن‌ها (مثل کارت‌های جمع‌شده و پرچم‌ها) رو برمی‌گردونه.
- نتیجه رو به صورت وضعیت شبیه‌سازی شده برمی‌گردونه.

```

● ● ●

def minimax(cards, player1, player2, depth, is_maximizing, alpha=-inf, beta=inf,
no_heuristic=False):
    """
    Uses the Minimax algorithm with Alpha-Beta pruning and optional deep search
    to determine the best move for a given game state.

    Parameters:
        cards (list): Represents the current game board as `Card` objects.
        player1 (Player): Player 1's state including banners and card
        player2 (Player): Player 2's state including banners and card
        depth (int): Depth limit for game tree exploration
        is_maximizing (bool): Indicates if the current player is maximizing the score.
        alpha (float): Alpha value for pruning (best score for maximizing player).
        beta (float): Beta value for pruning (best score for minimizing player).
        no_heuristic (bool, optional): Flag for additional end-game evaluation when
        no moves remain.

    Returns:
        tuple: Contains
        s:           - int: The highest or lowest score achievable from this state.
        s:           - int or None: The index of the optimal move, or None if no valid moves
        are available.
    """
    valid_moves = get_valid_moves(cards)

    if depth == 0 or len(valid_moves) == 0:
        if no_heuristic:
            if calculate_winner(player1, player2) == 1:
                winner_score = 2000
            else:
                winner_score = -2000
            return (winner_score, None)
        else:
            return (evaluate_state(cards, player1, player2), None)

    optimal_move = None
    if is_maximizing:
        max_eval = -inf
        for move in valid_moves:
            next_cards, next_player1, next_player2 = simulate_move(
                cards, player1, player2, move, player=1)
            current_eval, _ = minimax(next_cards, next_player1, next_player2, depth
            - 1, False, alpha, beta, no_heuristic)
            if current_eval > max_eval:
                max_eval = current_eval
                optimal_move = move
                alpha = max(alpha, current_eval)
            if beta <= alpha:
                break # Stop exploring this branch
        return max_eval, optimal_move
    else:
        min_eval = inf
        for move in valid_moves:
            next_cards, next_player1, next_player2 = simulate_move(
                cards, player1, player2, move, player=2)
            current_eval, _ = minimax(next_cards, next_player1, next_player2, depth
            - 1, True, alpha, beta, no_heuristic)
            if current_eval < min_eval:
                min_eval = current_eval
                optimal_move = move
                beta = min(beta, current_eval)
            if beta <= alpha:
                break # Stop exploring this branch
        return min_eval, optimal_move

```

الگوریتم مینیمکس با هرس آلفا-بتا (تابع minimax)

این تابع از یه الگوریتم جستجو استفاده می کنه تا بهترین حرکت رو پیدا کنه:

1. پایه‌ی بازگشتنی:

اول از همه، بررسی می کنه که عمق جستجو تموم شده یا حرکت دیگه‌ای باقی نمونده. اگه تموم شده باشه، وضعیت فعلی بازی رو با تابع evaluate_state امتیازدهی می کنه و برمی‌گردونه.

2. اگه نوبت بازیکن ما (Maximizing) باشه:

- تابع همه حرکت‌های ممکن رو بررسی می کنه.
- هر حرکت رو با تابع simulate_move شبیه‌سازی می کنه تا وضعیت جدید بازی رو به دست بیاره.

- بعد، به صورت بازگشتنی Minimax رو برای حریف (Minimizing) صدا می زنه.
- بهترین امتیاز رو نگه می داره و بقیه رو دور می ریزه.
- از هرس آلفا-بتا استفاده می کنه تا حرکت‌هایی که بررسی شون بی‌فایده‌ست رو رد کنه و سرعتش بره بالا.

3. اگه نوبت حریف (Minimizing) باشه:

- دقیقاً مثل مرحله بالا عمل می کنه، ولی هدفش اینه که بدترین امتیاز ممکن رو برای بازیکن ما انتخاب کنه.
- باز هم با هرس آلفا-بتا، حرکت‌های غیرضروری رو رد می کنه.

4. برگشت:

- بعد از بررسی همه حرکت‌ها، تابع بهترین امتیاز و حرکت مربوط به اون امتیاز رو برمی‌گردونه.

```
def get_move(cards, player1, player2):
    """
    Determines the best move for Player 1 using the Minimax algorithm with Alpha-Beta
    pruning.

    Parameters:
        cards (list): A list of `Card` objects representing the current game state.
        player1 (Player): The Player 1 object including banners and cards.
        player2 (Player): The Player 2 object including banners and cards.

    Returns:
        int or None: The index of the best move for Player 1, or None if no moves are
        available.
    """
    # if there is no limit in time:

    if len(cards) < 25:
        depth = 9
    else:
        depth = 5

    flag = len(cards) <= 16

    ...
    if len(cards) < 30 and len(cards) > 22:
        depth =
5        flag = False
    elif len(cards) <= 22 and len(cards) > 1
6        depth =
7        flag = False
    elif len(cards) <= 16 :
        depth =
9        flag = True
    else:
        depth =
3        flag = False
    ...

    best_move = minimax(
        cards=cards,
        player1=player1,
        player2=player2,
        depth=depth,
        is_maximizing=True,
        alpha=-inf,
        beta=inf,
        no_heuristic=flag
    )[1]

    return best_move
```

این تابع آخرین مرحله‌ایه که ایجنت تصمیم می‌گیره "الان باید چه حرکتی کنم؟". با کمک الگوریتم Minimax و تنظیمات خاصی که براساس وضعیت فعلی بازی انانجام می‌شه، بهترین حرکت ممکن رو برای بازیکن اول (ایجنت هوشمند ما) پیدا می‌کنه.

1. بررسی تعداد کارت‌های باقی‌مونده:

- تابع اول نگاه می‌کنه چندتا کارت روی تخته باقی مونده. تعداد کارت‌ها تعیین می‌کنه که عمق جستجوی Minimax چقدر باشه:
- کارت‌های زیاد: عمق کمتر (مثلاً 5) چون بررسی همه‌ی حالت‌ها زمان بر می‌شه.
- کارت‌های کم: عمق بیشتر (مثلاً 9) چون بازی داره به انتهای نزدیک می‌شه و بررسی دقیق‌تر لازمه.

2. فعال‌سازی حالت :no heurisit

- وقتی تعداد کارت‌ها خیلی کم می‌شه (مثلاً کمتر از 16 کارت)، یه حالت خاص به اسم "no heuristic" فعال می‌شه. این حالت به جای اینکه بخوایم از تابع هیوریستیک استفاده کنیم مستقیماً از خود استیت نهایی بازی استفاده می‌کنیم برای اینکه مشخص کنیم کی بازیو برده یا باخته.

3. اجرای Minimax

- بعد از تنظیم عمق و حالت جستجو، تابع minimax رو صدا می‌زنه و ارزش می‌خواهد که بهترین حرکت رو بر اساس وضعیت فعلی بازی پیدا کنه.

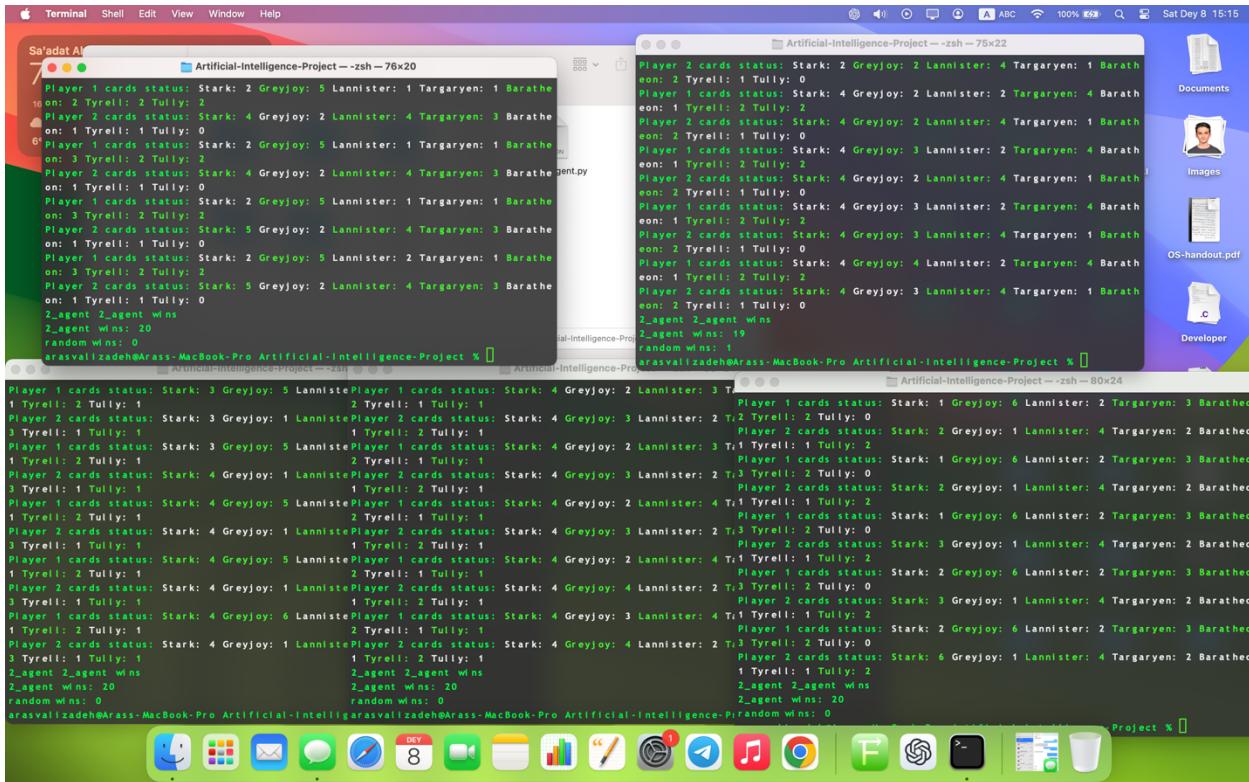
4. برگشت بهترین حرکت:

- وقتی Minimax نتیجه رو داد، این تابع شماره‌ی بهترین حرکت رو برمی‌گردونه تا ایجنت حرکتش رو انانجام بده.

نتائج:

در نهایت ۱۰۰ بار ایجنت خودمون رو با ایجنت رندهم بازی دادیم که موفق شد ۹۹ بار ایجنت رندهم رو شکست بده و فقط ۱ بار ببازه. هرچی عمق مینیماکس رو بیشتر کنیم طبعا درصد موفقیت بیشتر میشه ولی از اونور زمان اجرای بازی هم طولانی میشه.

اسکرین شات نتایج خروجی رو پایین قرار میدم.



در فاز دوم برای بهبود تابع هیورستیک از شبکه عصبی و ژنتیک استفاده خواهیم کرد که در ادامه توضیح داده خواهد شد.

مدل شبکه عصبی مورد استفاده از یک شبکه پرسپترون چندلایه (MLP) تشکیل شده که دارای ۴ لایه کاملاً متصل (Fully Connected) است:

لایه اول: شامل ۵۶ نورون (ورودی)، متصل به یک لایه مخفی با ۲۵۶ نورون

لایه دوم: دارای ۶۴ نورون

لایه سوم: دارای ۸ نورون

لایه خروجی: دارای ۱ نورون (نمایش امتیاز وضعیت بازی)

```
self.fc1 = nn.Linear(56, 256)
self.fc2 = nn.Linear(256, 64)
self.fc3 = nn.Linear(64, 8)
self.fc4 = nn.Linear(8, 1)
self.initialize_weights()
```

در این مدل، از تابع فعال‌سازی Leaky ReLU برای افزایش دقیق و جلوگیری از مشکل vanishing gradient استفاده شده است.

نرمال‌سازی ورودی‌ها

برای افزایش پایداری آموزش، ورودی‌های شبکه ابتدا استانداردسازی (Standardization) می‌شوند:

مقادیر ورودی‌ها میانگین‌گیری و نرمال‌سازی می‌شوند تا مدل بتواند بهتر تعمیم دهد.

مقادیر میانگین (mean) و انحراف معیار (std) در هنگام آموزش ذخیره می‌شوند و هنگام پیش‌بینی نیز روی داده‌ها اعمال می‌شوند.

```
__init__(self, input_mean, input_std):
super(MyModel, self).__init__()
self.input_mean = torch.tensor(input_mean, dtype=torch.float32) # Store mean
self.input_std = torch.tensor(input_std, dtype=torch.float32) # Store std
```

داده‌های آموزشی از فایل merged_dataset2.pt بارگیری می‌شوند که شامل:

ویژگی‌های ورودی (input_data): نمایشی از وضعیت بازی

خروجی (output_data): مقدار امتیاز وضعیت که عامل باید یاد بگیرد

پس از بارگیری، داده‌ها به صورت آرایه‌های NumPy تبدیل و استانداردسازی می‌شوند:

```
input_data = np.asarray(x['input_data'], dtype=np.float32)
output_data = np.asarray(x['output_data'], dtype=np.float32)
```

یک زمانبندی کاهش نرخ یادگیری (Learning Rate Scheduler) اضافه شده تا هر ۴۰۰ epoch، نرخ یادگیری کاهش یابد. این کار به همگرایی بهتر مدل کمک می‌کند.

(scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=400, gamma=0.3

مدل به مدت ۲۰۰۰ epoch با دسته‌های ۵۰۰ نمونه‌ای (Batch size = 500) آموزش داده می‌شود:

```
for epoch in range(epochs):
    epoch_loss = 0
    for inputs, targets in dataloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()          Aras Valizadeh, 7 hours ago • heuristic function
        optimizer.step()
        epoch_loss += loss.item()

    scheduler.step()

    epoch_loss /= len(dataloader)
    print(f"Epoch {epoch+1}/{epochs}, Avg Loss: {epoch_loss:.4f}")
```

پس از اتمام آموزش، مدل ذخیره شده و اطلاعات میانگین و انحراف معیار ورودی‌ها نیز ذخیره می‌شود:

```
torch.save({  
    'model_state_dict': model.state_dict(),  
    'input_mean': input_mean,  
    'input_std': input_std  
} ("best_model_with_norm9.pth")
```

نحوه استفاده از شبکه عصبی در عامل بازی

در کد عامل هوشمند، از این مدل برای ارزیابی وضعیت‌های مختلف بازی و انتخاب بهترین حرکت استفاده شده است. برای این کار، وضعیت بازی به بردار عددی تبدیل شده و به شبکه عصبی ارسال می‌شود:

```
state_tensor = torch.stack(representation([cards], [player1], [player2],  
[companion_cards])).float()  
  
with torch.no_grad():  
  
score = model(state_tensor)
```

نتایج و تحلیل عملکرد

۱. مزایا:

استفاده از شبکه عصبی، دقت ارزیابی وضعیت بازی را نسبت به روش‌های ساده‌تر افزایش داده است.

این روش باعث می‌شود که عامل سریع‌تر وضعیت‌های پیچیده را پردازش کند.

تکیب یادگیری نظارت شده (Supervised Learning) با جستجوی Minimax باعث شده که مدل هم از تجربه یاد بگیرد و هم قدرت جستجوی دقیقی داشته باشد.

۲. معایب:

- زمان آموزش مدل طولانی است.
 - ممکن است مدل دچار overfitting شود که باید با روش های بیشتری مانند Dropout یا افزایش داده ها (Data Augmentation) حل شود.
- جمع بندی و بهبودهای آینده

برای بهبود مدل، پیشنهادات زیر مطرح می شود:

1. استفاده از یادگیری تقویتی (Reinforcement Learning):
 - به جای یادگیری از داده های ثابت، می توان عامل را در برابر خودش بازی داد و مدل را بهبود بخشید.
2. اضافه کردن مکانیزم Dropout:
 - این کار به افزایش تعمیم پذیری مدل کمک می کند و از overfitting جلوگیری می کند.
3. افزایش داده های آموزشی (Data Augmentation):
 - اگر داده ها کافی نباشند، می توان با روش هایی مانند تولید وضعیت های جدید بازی، مجموعه داده را گسترش داد.

در این پژوهه، برای بهبود عملکرد عامل هوشمند در بازی "دست پادشاه" از روش ترکیبی (**Ensemble Method**) استفاده شده است. روش های ترکیبی با ترکیب چندین الگوریتم یادگیری، سعی در افزایش دقت پیش بینی و کاهش خطاهای مدل دارند. در این پژوهه، ما الگوریتم **Minimax** با هرس آلفا- بتا را با یک شبکه عصبی عمیق (**Neural Network**) ترکیب کردایم تا بهترین حرکت را در بازی انتخاب کنیم.

چرا از روش ترکیبی استفاده کردیم؟

هر الگوریتم به تنها دارای نقاط قوت و ضعف خاص خود است:

• یک الگوریتم کلاسیک برای جستجوی بهینه در فضای بازی است که تضمین می‌کند بهترین حرکت ممکن را انتخاب کند.

• مزیت: دقیق و بهینه در بازی‌های کامل

• عیب: با افزایش پیچیدگی بازی، زمان پردازش زیاد می‌شود.

• شبکه عصبی (Neural Network) می‌تواند از داده‌های قبلی یاد بگیرد و وضعیت‌های بازی را ارزیابی کند.

• مزیت: یادگیری از تجربه و کاهش پیچیدگی جستجو

• عیب: نیاز به داده‌های آموزشی و تنظیمات دقیق دارد.

```
def representation(full_cards: Card , player1s:Player , player2s:Player, companion_cards:Card):
    map_house = {'Stark': 1, 'Greyjoy': 2, 'Lannister': 3, 'Targaryen': 4, 'Baratheon': 5, 'Tyrell': 6, 'Tully': 7}
    map_companion_cards = {"Jon": 0,"Jaqen": 1,"Gendry": 2,"Melisandre": 3,"Ramsay": 4,"Sandor": 5}
    # making the representation for the cards
    ans = []
    for i in range(len(full_cards)):
        representation = torch.zeros((56))

        cards = full_cards[i]
        # print(cards)
        for card in cards:
            if card.name == "Varys":
                representation[card.location] = -1
                continue
            house_label = map_house[card.get_house()]
            house_location = card.location
            representation[house_location] = house_label
        # making the representation for the companion_cards
        cards = companion_cards[i]
        for card in cards:
            representation[36+map_companion_cards[card]] = 1
        # making the representation for the player1
        player1 = player1s[i]
        for key in map_house.keys():
            representation[41 + map_house[key]] = len(player1.cards[key])
        # making the representation for the player2
        player2 = player2s[i]
        for key in map_house.keys():
            representation[48 + map_house[key]] = len(player2.cards[key])
        ans.append(representation)
    return ans
```

Aras Valizadeh, 7 hours ago • heuristic function replaced with neural network

کد ارائه شده مسئول بارگذاری مدل شبکه عصبی آموزش دیده و تبدیل وضعیت بازی به نمایش عددی برای ارزیابی است. در ابتدا، مسیر فایل مدل (best_model_with_norm4.pth) مشخص شده و وزن‌های

ذخیره شده به همراه مقادیر میانگین (input_mean) و انحراف معیار (input_std) بارگذاری می‌شوند. سپس مدل نمونه‌سازی و در حالت ارزیابی (model.eval()) قرار داده می‌شود تا بتواند بدون بهروزرسانی وزن‌ها پیش‌بینی انجام دهد. تابع representation برای تبدیل وضعیت بازی به یک بردار ۵۶ بعدی تعریف شده است. این نمایش شامل موقعیت کارت‌های شخصیت، وضعیت کارت‌های همراه، و تعداد کارت‌های هر خاندان برای دو بازیکن است. این نمایش عددی، ورودی شبکه عصبی را تشکیل می‌دهد و مدل را قادر می‌سازد وضعیت بازی را ارزیابی کرده و به انتخاب بهترین حرکت کمک کند.