Scacchi - IA homework (Leonardo Serilli)

Lo scopo del progetto è stato quello di implementare una partita a scacchi tramite minmax con alpha betha pruning e successivamente per mezzo di un regressore lineare allenato con le partite giocate dalla minmax, in grado di predirre ciò che la minmax avrebbe calcolato a una data depth.

Il **dataset** è stato quindi **rimepito precedentemente** al codice descritto in questo report. Questo è stato riempito eseguendo ripetutamente partite tramite minmax con diverse combinazioni della **depth e euristiche di valutazione** tra quelle descrite nella sezione relativa al file `chess_tools.py`.

Inoltre nella versione attuale è giocata una partita tra **il Bianco, che utilizza il regressore, e il nero che utilizza la minmax search**.
Inoltre, in ogni partita giocata ogni mossa del giocatore nero viene aggiunta al dataset, ampliando la conoscenza che il bianco utilizzera nell'esecuzione successiva, infatti il **training del modello avviene all'inizio di ogni partita.**

I file python presenti sono descritti in seguito:

main.py

Il file **main.py** specifica alcune variabili necesarie al gioco, renderizza la scacchiera tramite un banale uso della **libreria pygame** e i print di informazioni utili riguardanti l'esecuzione.

Compreso tra le righe `132-142` è presente il codice per la gestione dei giocatori: **la mossa del bianco è scelta tramite minmax** e quella **del nero tramite regressore**.

Inoltre è visibile com ogni **mossa del nero** venga **registrata nel dataset**, in modo da essere disponibile per il regressore alla partita successiva.

```
132 _
133             if not(board.turn):
134                 # codice per giocare tramite la minmax e ampliare il dataset
135                 old_board = copy.deepcopy(board)
136                 board = minmax_ab(board, DEPTH, heuristics_ids) # esegue la mossa scelta dalla minmax search con alpha beta pruning
137                 add_sample_to_dataset(old_board, board, DEPTH, heuristics_ids)
138 
139             else:
140                 # codice per giocare tramite il regressore
141                 next_mv = regression(regressor, board, DEPTH)
142                 board.push_uci(str(next_mv))
```

chess_tools.py

Il file **chess_tools.py** contiene alcune funzioni utili sia alla minmax che al regressore:

La funzione **make_matrix(board)** trasforma la board in input in una matrice di caratteri:

```
6  # Ritorna una matrice dall'oggetto "board" della libreria chess
7  def make_matrix(board): #type(board) == chess.Board()
8      pgn = board.epd()
9      foo = []  #Final board
10     pieces = pgn.split(" ", 1)[0]
11     rows = pieces.split("/")
12     for row in rows:
13         foo2 = []  #This is the row I make
14         for thing in row:
15             if thing.isdigit():
16                 for i in range(0, int(thing)):
17                     foo2.append('.')
18             else:
19                 foo2.append(thing)
20         foo.append(foo2)
21     return foo
```

La funzione **board_eval(board, heuristics_ids)** valuta la board tramite le euristiche di valutazione specificate tramite un identificatore passato in input;

Le euristiche sono le seguenti:

- `0` **Conteggio dei pezzi**: questa è sempre utilizzata nella valutaione, il suo scopo è sommare i valori associati a ogni pezzo presente nella board

	Bianco	Nero
Pedone	1	-1
Alfiere	3	-3
Cavallo	3	-3
Torre	5	-5
Regina	9	-9
Re	0	0

- `1` **Massimizzo controllo scacchiera**: la valutazione migliora in base al numero di mosse possibili che si possino eseguire
- `2` **Scacco al Re**: la valutazione migliora se il re avversario viene messo sotto scacco da una tua mossa

```python
def boardEval(board, heuristics_ids):

    score = 0
    piece_values = {'p': -1, 'b': -3, 'n': -3, 'r': -5, 'q': -9, 'k': -0,
            'P': 1, 'B': 3, 'N': 3, 'R': 5, 'Q': 9, 'K': 0}

    # migliora lo score rispetto il controllo che si ha sulla scacchiera
    if(1 in heuristics_ids):
        if(board.turn):
            score += 10*board.legal_moves.count()
        else:
            score -= 10*board.legal_moves.count()

    mvs = [str(mv)[:4] for mv in board.legal_moves]
    turn = board.turn
    opp_k = ""
    board = make_matrix(board)

    # lo score migliora proporzionatamente al numero di pezzi che si hanno in più rispetto l'avversario
    for i in range(len(board)):
        for j in range(len(board[i])):
            if(board[i][j] != '.'):
                score += piece_values[board[i][j]]

    # lo score migliora se il re avversario viene messo sotto scacco
    if(2 in heuristics_ids):
        for i in range(len(board)):
            for j in range(len(board[i])):
                if(turn):
                    if(board[i][j] == 'K'):
                        opp_k = str(MAPPING[i]) + str(j)
                    for mv in mvs:
                        if(opp_k == mv[2:]):
                            score += 0.8
                else:
                    if(board[i][j] == 'k'):
                        opp_k = str(MAPPING[i]) + str(j)
                    for mv in mvs:
                        if(opp_k == mv[2:]):
                            score -= 0.8
    return score
```

minmax_alpha_betha.py

il file **minmax_alpha_betha.py** contiene la funzione ricorsiva per la **minmax search con alpha/betha pruning**.

```python
def minmax_search_ab(node, a, b, depth, best_mv, heuristic_ids):

    #condizione di base
    if(depth == 0 or node.is_checkmate()):
        return {"node": node, "eval": boardEval(node, heuristic_ids)}

    # turno del maximizer
    if(node.turn):
        maxEval = -math.inf
        for child in childs(node):
            elem = minmax_search_ab(child, a, b, depth-1, best_mv, heuristic_ids)
            evaluation = elem["eval"]
            maxEval = max(maxEval, evaluation)

            # aggiorna ricorsivamente la configurazione migliore a cui una mossa del maximizer può portare
            if(maxEval == evaluation):
                best_mv = elem["node"]

            # alpha-beta pruning
            a = max(a, evaluation)
            if(b <= a):
                break
        return {"node": node, "eval": maxEval, "best-child": best_mv}
    # turno del minimizer
    else:
        minEval = math.inf
        for child in childs(node):
            elem = minmax_search_ab(child, a, b, depth-1, best_mv, heuristic_ids)
            evaluation = elem["eval"]
            minEval = min(minEval, evaluation)

            # aggiorna ricorsivamente la configurazione migliore a cui una mossa del minimizer può portare
            if(minEval == evaluation):
                best_mv = elem["node"]

            # alpha-beta pruning
            b = min(b, evaluation)
            if(b <= a):
                break
        return {"node": node, "eval": minEval, "best-child": best_mv}
```

regressor.py

il file **regressor.py** contiene le funzioni necessarie per la crezazione del dataset, la creazione del regressore e per il suo utilizzo.

Il **dataset** utilizzato dal regressore ha la seguente forma e contiene una **codifica in numeri interi delle feature**, e la valutazione dello stato della scacchiera:

	a1	b1	c1	d1	e1	f1	g1	h1	a2	b2	c2	d2	e2	f2	g2	h2	a3	b3	c3	d3	e3	f3	g3	h3	...	d6	e6	f6	g6	h6	a7	b7	c7	d7	e7	f7	g7	h7	a8	b8	c8	d8	e8	f8	g8	h8	depth	eval	next_mv
0	3	5	7	9	11	7	5	3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	2	2	2	2	2	2	4	6	8	10	12	8	6	4	3	200.0	1124
1	3	5	7	9	11	7	5	3	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	2	0	2	2	2	2	4	6	8	10	12	8	6	4	3	-200.0	390
2	3	5	7	9	11	7	5	3	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	220.0	66
3	3	5	7	9	11	7	5	3	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	-200.0	2506
4	3	5	7	9	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	210.0	2182
5	3	5	7	9	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	0	2	2	2	4	6	8	10	12	8	6	4	3	-290.0	2075
6	3	5	7	0	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	0	2	2	2	4	6	8	10	12	8	6	4	3	240.0	1653
7	3	5	7	0	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	0	0	2	2	2	4	6	8	10	12	8	6	4	3	-400.0	919
8	3	5	7	9	11	7	5	3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	2	2	2	2	2	2	4	6	8	10	12	8	6	4	3	200.0	1124
9	3	5	7	9	11	7	5	3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	-200.0	390
10	3	5	7	9	11	7	5	3	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	220.0	66
11	3	5	7	9	11	7	5	3	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	-200.0	2506
12	3	5	7	9	11	7	5	3	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	2	2	2	2	2	2	4	6	8	10	12	8	6	4	3	200.0	1124
13	3	5	7	9	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	210.0	2182
14	3	5	7	0	11	7	5	3	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	0	0	2	2	2	4	6	8	10	12	8	6	4	3	350.0	1256
15	3	5	7	9	11	7	5	3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	-200.0	390
16	3	5	7	9	11	7	5	3	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	2	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	220.0	66
17	3	5	7	9	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	0	2	2	2	4	6	8	10	12	8	6	4	3	-290.0	2075
18	3	5	7	9	11	7	5	3	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	2	2	2	2	4	6	8	10	12	8	6	4	3	-200.0	2506
19	3	5	7	0	11	7	5	3	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	2	0	2	0	2	2	2	4	6	8	10	12	8	6	4	3	240.0	1653

Le variabili `MAPPING` e `CHESSBOARD` sono utilizzate per mappare in numeri interi mosse e posizioni dei pezzi.

```
MAPPING = {".":0, "p":1, "P":2, "r":3, "R":4, "n":5, "N":6, "b":7, "B":8, "q":9,
"Q":10, "k":11, "K":12}
CHESSBOARD = ["a1", "a2", "a3", "a4", "a5", "a6", "a7", "a8", "b1", "b2", "b3",
"b4", "b5", "b6", "b7", "b8", "c1", "c2", "c3", "c4", "c5", "c6", "c7", "c8",
"d1", "d2", "d3", "d4", "d5", "d6", "d7", "d8", "e1", "e2", "e3", "e4", "e5",
"e6", "e7", "e8", "f1", "f2", "f3", "f4", "f5", "f6", "f7", "f8", "g1", "g2",
"g3", "g4", "g5", "g6", "g7", "g8", "h1", "h2", "h3", "h4", "h5", "h6", "h7",
"h8"]
```

Le funzioni **gen_moves()**, **encode_mv** e **decode_mv()** sono utilizzate per il mapping delle mosse, espresse dalla libreria `chess` come stringe di 4 o 5 caratteri, in interi, in modo da poterli aggiungere a un sample del dataset per poter poi essere utilizzati dal modello senza bisogno di post processing del dataset.

```
# genera un array dell possibili mosse sulla scacchiera
def gen_moves():
    moves = []
    for i in CHESSBOARD:
        for j in CHESSBOARD:
            if(i!=j):
              moves.append(i+j)
            if((i[1] == '2' and j[1] == '1') or (i[1] == '7' and j[1] == '8')):
                moves.append(i+j+'q')
    return moves

# utilizza la posizione nell'array generato da gen_moves() per la codifica delle mosse
def encode_mv(mv):
    moves = gen_moves()
    if(len(str(mv))>4):
            mv = str(mv)[:4]+'q'
    return moves.index(str(mv))

# decodifica una mossa utilizzando l'array generato da gen_moves()
def decode_mv(enc_mv):
    moves = gen_moves()
    while(enc_mv > len(moves)-1):
        enc_mv -= 1
    return moves[enc_mv]
```

ampliassero il datasetLa funzione **create_sample(board, depth, heuristics_ids)** crea un sample dalla boarda attuale, senza specificare la prossima mossa, poichè il sample qui generato è poi dato in input al regressore che predirrò la mossa.

```
# crea un sample dalla board attuale, senza specificare la prossima mossa_
def create_sample(board, depth, heuristics_ids):
    sample = []
    evaluation = boardEval(board, heuristics_ids)
    next_mv = ""
    board = make_matrix(board)

    for i in range(len(board)):
        for j in range(len(board)):
            sample.append(MAPPING[board[i][j]])

    sample.append(depth)
    sample.append(evaluation)
    return sample
```

La funzione **add_sample_to_dataset(old_board, board, heuristics_id)** viene chiamata ogni volta che una mossa è effettuata tramite minmax, e prende come input sia la configurazione della board prima della mossa sia dopo.

Il codice **genera un sample e lo appende al dataset** tramite la funzione **append_to_csv(sample)** che semplicemente aggiunge la riga a un file `.csv` e se non esiste lo crea.

```python
# crea un sample dalla board attuale, senza specificare la prossima mossa_
def create_sample(board, depth, heuristics_ids):
    sample = []
    evaluation = boardEval(board, heuristics_ids)
    next_mv = ""
    board = make_matrix(board)

    for i in range(len(board)):
        for j in range(len(board)):
            sample.append(MAPPING[board[i][j]])

    sample.append(depth)
    sample.append(evaluation)
    return sample
```

La funzione **model()** *crea, allena e valuta il regressore lineare*. Tra le informazioni stampate c'è la **radice del MSE** ottenuta dalla valutazione del modello sul train set.

```python
# crea, allena e valuta il regressore
def model():
    columns = []
    for i in range(1,9):
        for j in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']:
            columns.append(j+str(i))

    columns.append("depth")
    columns.append("eval")
    columns.append("next_mv")


    df = pd.read_csv("dataset.csv", names=columns)
    print("\n", df)
    y = df["next_mv"]
    x = df.drop(["next_mv"], axis=1)
    x = x.apply(pd.to_numeric, errors='coerce')
    y = y.apply(pd.to_numeric, errors='coerce')
    TS_x, tst_x, TS_y, tst_y = train_test_split(x, y, test_size=0.2, train_size=0.8, shuffle=True)

    print("\nTS_x shape: ", TS_x.shape)
    print("TS_y shape: ", TS_y.shape)
    print("tst_x shape: ", tst_x.shape)
    print("tst_y shape: ", tst_y.shape)

    print("\nTraining regressor")
    regressor = LinearRegression().fit(TS_x, TS_y)
    print("training ended")

    predictions = regressor.predict(tst_x)
    score = mean_squared_error(predictions, tst_y)

    return regressor
```

La funzione **regression(regressor, board, depth)** ritorna il valore predetto dal sample associato alla configurazione attuale della board e della depth che utilizzerebbe la minmax al suo posto.

La mossa predetta viene rimappata nel formato accettato dalla libreria `chess`.

Per via delle imprecisioni sulla predizione, questa può non essere presente tra le mosse legali, di conseguenza viene calcolata **la mossa legale che più si avvicina alla predizione**, e questa viene utilizzata dal giocatore.

```
145 # computa la mossa legale che più si avvicina alla predizione
146 def nearest_legal_mv(mv, board):
147     nearest = None
148     dist = float('inf')
149     for el in board.legal_moves:
150         if(abs(encode_mv(el) - encode_mv(mv))<dist):
151             nearest = el
152             dist = abs(encode_mv(el) - encode_mv(mv))
153     return nearest
```

La funzione **nearest_legal_mv(mv, board)** ritorna la mossa legale più vicina alla predizione.

```
# computa la mossa legale che più si avvicina alla predizione
def nearest_legal_mv(mv, board):
    nearest = None
    dist = float('inf')
    for el in board.legal_moves:
        if(abs(encode_mv(el) - encode_mv(mv))<dist):
            nearest = el
            dist = abs(encode_mv(el) - encode_mv(mv))
    return nearest
```

Conclusioni

Nei risultati ottenuti è stato possibile identificare una **velocita notevolmente superiore del regressore nel predire la mossa rispetto alla minmax**.

Seppur utilizzando la tecnica dell'**alpha/betha pruning**, per depth superiori a 3 non è possibile ottenere tempi ragionevoli.

Nonostante la velocità pessima però il giocatore che utlizza la **minmax search ha ottenuto la vittoria per scacco matto più spesso del regressore**, questo probabilmente dovuto a due fattori:

1. La semplicità del modello utilizzato ;
2. La grandezza del dataset insufficente.

Per far fronte al secondo punto ho lasciato , nelle partite del regressore contro il minmax, che le mosse del secondo **ampliassero il dataset**, cosi che il regressore, nelle successive esecuzioni, continui a migliorare il suo gioco.

È allegato un video che mostra un esecuzione del codice