Artificial Intelligence: Reinforcement Learning project

Reduced chess end-games

Edoardo Ghini

October 4, 2017

Dipartimento di Ingegneria dell'Università di Roma La Sapienza

Contents

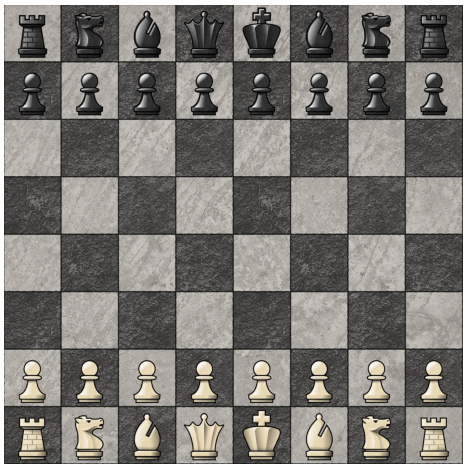


Figure 1: the game of chess

Part I

Introduction

A biologically inspired solution to solve the problem of autonomous learning is reinforcement learning.

This strategy tries to take advantage of the concept of reward and punishment that is present in nature.

In fact, it will stimulate an autonomous agent with positive or negative feedbacks after that each action has been accomplished, with the purpose of strengthen the agent's "decision criterion" which brings it to take the best among all the possible actions.

I have thought that one possible play-ground in which would be possible to experiment a reinforcement learning approach to a problem is, certainly, the widely known game of chess.

Part II

Development

1 Overview

The scenario chosen to develop the project is a chess game in a simplified environment.

In brief, two agents will play a chess game, one with a random behaviour, the other acting with the objective to maximise the reward given by the environment.

The autonomous agent, at first, will go through a learning phase using the Every Visit Monte Carlo paradigm.

Then it will act accordingly to highest expected value contained in the Q-table that was being populated during the learning phase.

Finally, all the performances will be harvested and arranged in plots that should allow to visualise the effectiveness of the reinforcement learning algorithm.

2 Implementative Solution

In this section I will describe the general structure of the project in a bottom-up fashion

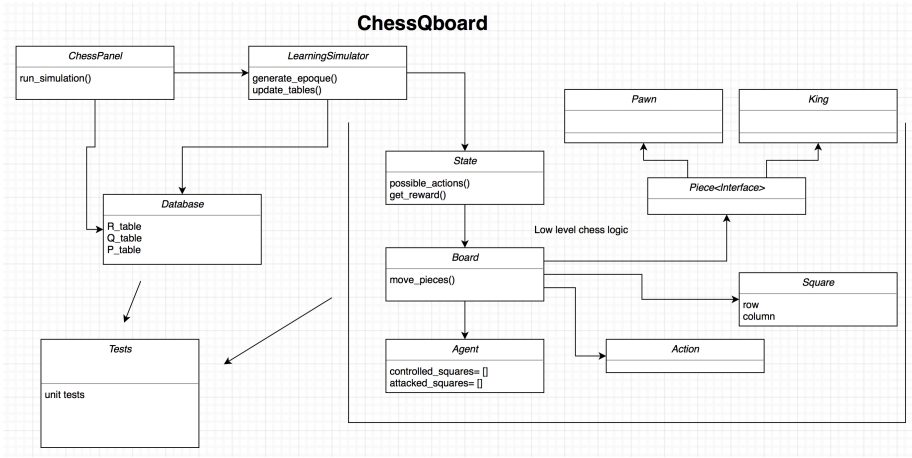


Figure 2: UML scheme of the project

2.1 Low Level Logic

I started from this part of the code, since, implementing the rules and logic of chess would give me a better understanding of the overall problem.

Python language was chosen because of the great readability of the code. An object oriented paradigm allows to divide the responsibilities of all the low logic to the objects which they belongs.

All the classes has been created from scratch.

At first, the wrapper that should act as a bridge between low and high levels is the State class which embodies all the properties of the state of the system, in this case it describes all the chess pieces that are on the chessboard.

After that, as should be clearer from the UML schema in fig(2), there is a Board class which accomplishes tasks like computing the possible actions at disposal of the agents.

In order to complete the low level overview there are also Agent, Piece, Square classes among others that helps to carry on high level tasks issued by State class, dividing each in micro tasks.

2.2 High Level Logic

Starting from the ChessPanel which contains the main calls, the execution flux will go in LearningSim class that probably contains the most meaningful part of the whole implementation.

In fact, in this class the learning process will be carried out. The function generate_epoque will obviously create a series of state action tuples (a Markov episode) which link the initial state to whatever final state encountered.

```
1  def generate_epoque(self,iteration_num,epoque_num):
2
3          step_number= 0
4          state = self.initial_state
5          turn= "white"
6          state_action_tuples= []
7          while step_number <= self.max_epoque_steps and not state.
               is_final_state_for_agent(turn):
8                  if turn == "white":
9                          possible_actions= state.possible_actions("
                               white")
10                 else:
11                         possible_actions= state.possible_actions("
                               black")
12
13                 #choose an action according to policy with epsilon-
                       greedy and softmax
14                 action=self.choose_action(state,possible_actions,
                       epoque_num,turn)
15                 next_state_str= state.execute_action( action )
16
17                 next_state= State.State(next_state_str)
18                 sa_tuple= [ state, action, turn]
19                 state_action_tuples.append( sa_tuple )
20
21                 state= next_state
22                 step_number+=1
23                 if turn == "white":
24                         turn= "black"
25                 else:
26                         turn= "white"
27         if state.is_final_state_for_agent(turn):
28                 state_action_tuples.append( [state, None, turn ])
29
30         return state_action_tuples
```

Code 1: generate epoque function

2.3 Monte Carlo Learning Theory

For this kind of problems, with huge state spaces that initially are all unknown, it could be convenient to sample some causal "from state, to state" chains from the state space in uniform way in order to partially explore the search space following a uniformly random distribution (at least for the first iterations).

These samples are called episodes and are at the base of Monte Carlo algorithm: after that this episodes have been generated, they could be used to update the policy.

I opted for a Monte Carlo Q function based update of the policy, instead of the version that uses V function. In fact, in retrospect, I think that I used the Q table itself both as concrete representation of the Q function and as the current policy used to decide the best next action. I may be correct in this, since the actual policy gives by definition the action that maximise the current Q function and similarly I choose the action that shows the highest value of expected reward in the actual Q table.

Finally, I will seek convergence through the update of the aforementioned Q table.

2.4 Database Integration

At a certain point I had to address the problem of data structure resilience, I wanted to keep in memory three different tables that are necessary for Monte Carlo algorithm and performance tracking.

Then I opted for usage of a database (MySql) that will help also in retrieving some useful informations such the most promising action from a particular state of the chessboard.

```
1  SELECT ACTION FROM QTABLE WHERE STATE="1k1/ppp/111/111/PPP/1K1" AND
       TURN="white" AND REWARD IN ( SELECT MAX(REWARD) FROM QTABLE
       WHERE STATE="1k1/ppp/111/111/PPP/1K1" AND TURN="white")
```

Code 2: db query for best action

This solution turned to be efficient in terms of data separation and it showed to be extremely to retrieve particular max or average values when they was needed without impacting on the complexity of python code.

But, on the other hand, it has added a very high time overhead to the code execution since every iteration of the algorithm triggers hundreds of db queries.

Such a trade off should not be undertaken lightheartedly since the time needed to train the model against a scenario of moderate complexity would skyrocket with the increment of possible states in the agent world.

2.4.1 R and Q tables

Among the tables stored in the database there are two that are necessary to the Monte Carlo algorithm execution:

The R table contains all the actions that have been chosen by agents indexed by a progressive integer and consequently the state from which the agent has done the action, the collected reward and also informations about the current epoque and iteration as shown in fig(3).

ID	STATE	ACTION	REWARD	TURN	EPOQUE	ITERATION
1	1k1/ppp/111/111/PPP/1K1	K.a2-a1	-5	white	1	2
2	1k1/ppp/111/111/PPP/1K1	K.a2-c1	100	white	5	2
3	1k1/ppp/111/111/PPP/1K1	K.a2-a1	-100	white	2	4
4	1k1/ppp/111/111/PPP/1K1	K.a2-a1	-52	white	5	2
5	1k1/ppp/111/111/PPP/1K1	K.a2-c1	100	white	3	1

Figure 3: table of rewards in database

The Q table, in fig(4), instead is composed by a set of state action pairs executed in the past with relative reward. In this data table the unique key is made of state and action attributes that are two strings and they describes uniquely the corresponding python class object in the code.

STATE	ACTION	REWARD	TURN
1k1/p1p/111/111/P1P/1K1	p.a5-a4	80	black
1k1/p1p/111/111/P1P/1K1	p.c5-c4	80	black
1k1/ppp/111/111/PPP/1K1	K.a2-a1	-52.3333	white
1k1/ppp/111/111/PPP/1K1	K.a2-c1	100	white

Figure 4: Q table in database

Tanks to this strict correspondence I was able to transfer and memorise strings of states and actions via database and when they where needed in the algorithm execution, they would be parsed and converted in objects.

```
def parse_action_string( self, action_string, board):
        letters= list(action_string)
        from_square_key = letters[2]+letters[3]
        to_square_key = letters[5]+letters[6]
        self.piece= board.squares[from_square_key].piece
        self.target_square= board.squares[to_square_key]
        self.capture= letters[4] == "x"
        self.check= len(letters) == 8
```

Code 3: from string to class object

These are examples of the dual nature of the action and of the direct and inverse parsing function.

```
def __str__(self):
        if self.capture:
                sign_1= "x"
        else:
                sign_1= "-"

        if self.check :
                sign_2 = "+"
        else:
                sign_2 = ""

        return self.piece.__str__() +"."+ self.piece.square.__str__
            () + sign_1 + self.target_square.__str__() + sign_2
```

Code 4: inverse mapping

2.5 Unit Testing

Another technique, which I found extremely useful, was to keep bugs in the code under an acceptable level through tests, that will address every single function of the low level logic.

2.5.1 Test Suite

I grouped every written test in classes specular to the classes which at which they refer.

Then, executing them all at once, I was able to fight bug propagation and code regression (bugs in old code introduced by implementation of new code).

```
1  def test_get_winner(self):
2          self.assertEqual( "white", self.final_state_checkmate.
               get_winner())
3          self.assertEqual( "white", self.final_state_pawn_in_last_row
               .get_winner())
4          self.assertEqual( None, self.final_state_stalemate.
               get_winner())
5          self.assertEqual( None, self.drawn_state.get_winner())
6          self.assertEqual( None, self.initial_state.get_winner())
```

Code 5: one test in State.py

2.5.2 TDD

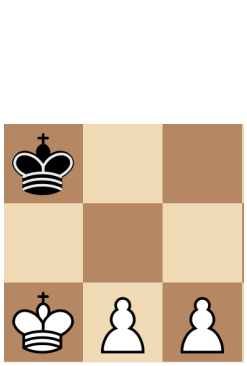
Generally, TDD (Test Driven Development) could be seen as work backwards, this is due to the fact that first the test of a function would be written, and indeed it will born to fail since the linked function is not implemented yet.

Then the function would be built with the assurance brought by the test that it will behave as expected.

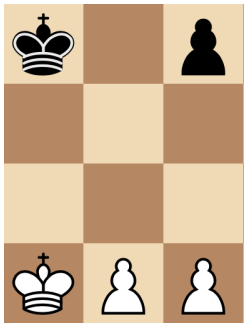
I found very reassuring and practical, to develop most of the low level code in this way.

2.6 Visualising Scenarios

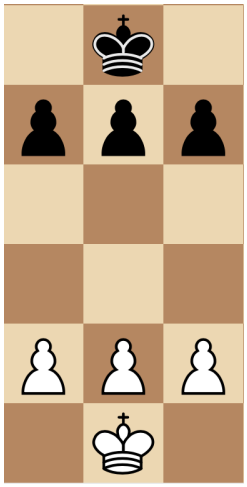
There I present some figures which can help to explain what is the problem addressed by the project.



(a) Simple board



(b) Moderated Board



(c) Complex Board

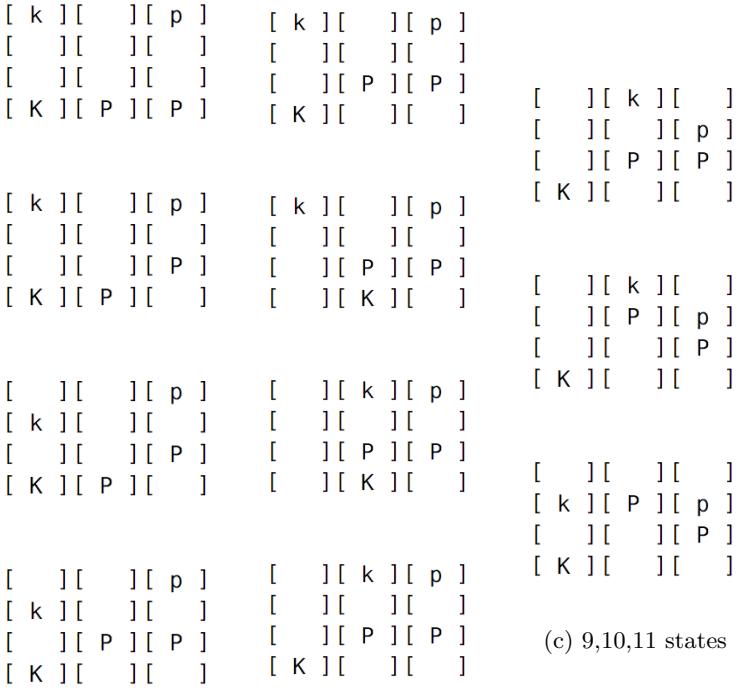
Even if the board configurations above seem trivial, there are a lot of possible sequences of states that can be generated from them, therefore this will reduce drastically the chances of convergence.

3 Results

After the learning process has come to an end, the statistics can be drawn from the table of performances that keeps record of the dimension of the Q table, of various averages of rewards and of the winner of the episode for each epoque in each iteration.

3.1 Displaying an episode

Since all the actions undertaken by each agent during episodes have been collected and indexed in the R table of the database, would be possible to display a sort of playback of the chess game.



(a) 1,2,3,4 states

(b) 5,6,7,8 states

(c) 9,10,11 states

Figure 6: Single game execution

As shown from the images above the state of the board is being modified by agents's actions.

3.2 Totally Greedy

An interesting possibility would be to exploit to the maximum the estimate of the Q value.

```
1  def run_some_greedy_games(self,games_num):
2          iteration_num= self.db.get_last_iteration_number()
3          iteration_num+=1
4          counter=1
5          for _ in range(games_num):
6
7                  self.simulator= ls.LearningSim(self.
                       initial_board_state ,self.max_epoque_steps,self.
                       agent_chosen,self.db,1,totally_greedy_flag=True)
8
9                  winner= self.simulator.montecarlo_epoque_exec(
                       iteration_num,counter)
10                 self.save_statistics(iteration_num,counter,winner)
11                 counter+=1
```

Code 6: greedy run function

This could be achieved by setting the epsilon parameter in a way in which the actions of the autonomous agent will be chosen always with exploitation in mind.

3.3 P table

As I have already mentioned, the P table stores all the statistics of the iterations and from there the performances could be evaluated.

ITERATION	EPOQUE	WAVGR	WAVGQ	BAVGR	BAVGQ	QSIZE	WINNER
1	1	63.5	63.64	-63.8	-63.64	11	white
1	2	0.67	25.81	-0.67	-23.14	29	NULL
1	3	-60	2.02	66.67	3.8	40	black
1	4	61.62	25.49	-61.58	-20.72	65	white
1	5	-0.75	22.97	0.67	-18.89	71	NULL
1	6	62.1	29.84	-62.11	-24.23	82	white
1	7	-0.67	27.61	0.6	-23.4	89	NULL
1	8	-0.75	26.03	0.67	-22.02	93	NULL
1	9	-0.6	23.68	0.75	-20.16	97	NULL
1	10	-60	18.42	68	-12.3	103	black

Figure 7: table of statistics

3.4 Experiments

3.4.1 Simple Case

At the beginning I tried with one of the simplest situations possible as shown in fig.

With this piece disposition there only the chance to win or stall for the autonomous agent, in other words there are only two possibilities, the learning agent ("white" in this case) wins against the random agent or the game ends by stalemate caused by poor choice of moves of learning agent.

The depth of the episodes is elementary: two actions at most. In fact the resulting tables after 300 epoques of learning and 30 of greedy playing are very small: 1302 entries in R table and 52 into the Q table.

ITERATION	EPOQUE	WAVGR	WAVGQ	BAVGR	BAVGQ	QSIZE	WINNER
1	1	-0.5	-0.73	1	0.73	3	NULL
1	2	62.62	54.19	-62.71	-60.4	16	white
1	3	-0.5	53.69	1	-59.68	16	NULL
1	4	-1	48.22	NULL	-59.68	17	NULL
1	5	-1	48.22	NULL	-59.68	17	NULL
1	6	68	55.02	-68	-64.2	18	white
1	7	0.6	43.39	-0.75	-50.01	21	white
1	8	-0.5	43.22	1	-49.65	21	NULL
1	9	-1	43.22	NULL	-49.65	21	NULL
1	10	-0.5	43.11	1	-49.39	21	NULL

Figure 8: beginning of the P table

From fig(8) is clear how the randomness of action choice due to the epsilon variable the learning agents often makes the wrong choice, however in fig(9) we can see how the agent will perform better in the final stages of learning and during the greedy games: it wins every time.

1	295	68	50.74	-68	-49.05	52	white
1	296	68	50.74	-68	-49.06	52	white
1	297	68	50.74	-68	-49.07	52	white
1	298	68	50.75	-68	-49.07	52	white
1	299	68	50.75	-68	-49.08	52	white
1	300	68	50.75	-68	-49.09	52	white
2	1	68	50.76	-68	-49.09	52	white
2	2	68	50.76	-68	-49.1	52	white
2	3	68	50.76	-68	-49.11	52	white
2	4	68	50.77	-68	-49.11	52	white
2	5	68	50.77	-68	-49.12	52	white
2	6	68	50.77	-68	-49.12	52	white
2	7	68	50.78	-68	-49.13	52	white
2	8	68	50.78	-68	-49.14	52	white
2	9	68	50.78	-68	-49.14	52	white
2	10	68	50.78	-68	-49.15	52	white

Figure 9: end of the P table

3.4.2 Moderate Case

After the simple case, I tried to solve a slightly more difficult scenario in which the potential outcome of the game was not as mono-directional as in the simple case. This increase of complexity is confirmed by the size of the new tables after the learning phase: this time R table counts 4606 entries and Q table grows up to 1305.

ITERATION	EPOQUE	WAVGR	WAVGQ	BAVGR	BAVGQ	QSIZE	WINNER
1	1	63.5	63.64	−63.8	−63.64	11	white
1	2	0.67	25.81	−0.67	−23.14	29	NULL
1	3	−60	2.02	66.67	3.8	40	black
1	4	61.62	25.49	−61.58	−20.72	65	white
1	5	−0.75	22.97	0.67	−18.89	71	NULL
1	6	62.1	29.84	−62.11	−24.23	82	white
1	7	−0.67	27.61	0.6	−23.4	89	NULL
1	8	−0.75	26.03	0.67	−22.02	93	NULL
1	9	−0.6	23.68	0.75	−20.16	97	NULL
1	10	−60	18.42	68	−12.3	103	black
1	11	−0.75	17.73	0.67	−11.74	106	NULL
1	12	−0.67	16.17	0.62	−10.86	115	NULL
1	13	64.4	19.59	−64.5	−14.15	121	white
1	14	−0.75	19.13	0.67	−13.37	123	NULL
1	15	−0.75	18.15	0.67	−12.82	128	NULL
1	16	0.62	16.71	−0.71	−11.75	140	white
1	17	−0.67	15.34	0.62	−10.53	147	NULL
1	18	62.62	19.73	−62.71	−15.27	159	white
1	19	62.1	23.94	−62.11	−20.19	174	white
1	20	−0.67	21.9	0.62	−18.38	190	NULL
1	21	−0.67	20.81	0.6	−17.44	196	NULL
1	22	−0.71	19.37	0.67	−16.41	205	NULL
1	23	−0.64	17.51	0.62	−14.98	225	NULL
1	24	−0.67	16.07	0.64	−13.69	245	NULL
1	25	0.67	15.73	−0.6	−13.38	251	white
1	26	−60	13.33	65	−9.04	259	black
1	27	63.5	15.09	−63.8	−10.09	263	white
1	28	64.4	16.45	−64.5	−11.19	266	white
1	29	62.62	18.95	−62.71	−13.85	279	white

(a) start of P table

1	300	65.5	14.81	−66	−9.65	1305	white
2	1	65.5	14.81	−66	−9.65	1305	white
2	2	64.4	14.81	−64.5	−9.65	1305	white
2	3	65.5	14.82	−66	−9.65	1305	white
2	4	65.5	14.82	−66	−9.65	1305	white
2	5	63.5	14.82	−63.8	−9.66	1305	white
2	6	63	14.82	−63.17	−9.66	1305	white
2	7	65.5	14.82	−66	−9.66	1305	white
2	8	64.4	14.82	−64.5	−9.67	1305	white
2	9	65.5	14.82	−66	−9.67	1305	white
2	10	65.5	14.82	−66	−9.67	1305	white
2	11	64.4	14.82	−64.5	−9.68	1305	white
2	12	64.4	14.83	−64.5	−9.69	1305	white
2	13	65.5	14.83	−66	−9.7	1305	white
2	14	0.6	14.83	−0.75	−9.69	1305	white
2	15	65.5	14.83	−66	−9.69	1305	white
2	16	65.5	14.83	−66	−9.69	1305	white
2	17	63.5	14.83	−63.8	−9.69	1305	white
2	18	64.4	14.83	−64.5	−9.7	1305	white
2	19	65.5	14.83	−66	−9.7	1305	white
2	20	62.62	14.83	−62.71	−9.7	1305	white
2	21	65.5	14.83	−66	−9.7	1305	white
2	22	63.5	14.83	−63.8	−9.7	1305	white
2	23	65.5	14.83	−66	−9.71	1305	white
2	24	65.5	14.84	−66	−9.71	1305	white
2	25	64.4	14.84	−64.5	−9.71	1305	white
2	26	65.5	14.84	−66	−9.71	1305	white
2	27	64.4	14.84	−64.5	−9.72	1305	white
2	28	65.5	14.84	−66	−9.72	1305	white
2	29	64.4	14.84	−64.5	−9.72	1305	white
2	30	63.5	14.84	−63.8	−9.72	1305	white

(b) end of P table

Figure 10: Performances of moderate case

Even if the possible state space is much more extended than the simple case, executing 300 learning epoques and 30 greedy games is sufficient to reach good performances as shown in fig and also the whole process is not very time expensive since it requires at most one minute.

3.4.3 Complex Case

In the end I tried to work with a complex scenario, obviously this would appear extremely simplified with the respect of the traditional chess board but it is certainly more complicated than simple and moderate cases that I had tried at that time.

From the starting position is shown in fig(5c), after some attempts of learning iterations with low numbers of epoques, checking the results I thought that the algorithm would require much more learning effort than before.

So I began to train the model on 5000 epoques and after that I execute 500 greedy games, the whole process kept going for several hours, almost 20.

pk1/111/P1p/11P/1K1/111	K.b2–b1	−1.25	white
pk1/111/P1p/11P/1K1/111	k.b6–c6	−3.4	black
pk1/111/P1P/1KP/111/111	P.a4–a5+	−0.725714	white
pk1/111/PK1/111/111/111	k.b6–c6	−82.9213	black
pk1/111/PpP/111/11P/11K	K.c1–b1	0.502222	white
pk1/111/PpP/111/11P/1K1	k.b6–a5	−0.52	black
pk1/11P/P11/111/111/1K1	k.b6xc5	−0.805405	black
pk1/11P/P11/111/11K/111	k.b6–c6	0.394393	black
pk1/11P/P11/111/11K/111	k.b6xc5	0.412717	black
pk1/1P1/111/1P1/111/K11	K.a1–b2	0.608889	white
pk1/1P1/111/1P1/1K1/111	p.a6–a5	0.626667	black
pk1/1P1/111/P1K/111/111	k.b6xb5	0.530578	black
pk1/1p1/111/P1P/11P/11K	P.c3–c4	0.231718	white
pk1/1p1/11P/P11/11P/11K	k.b6–c6	0.235242	black
pk1/1p1/11P/P1P/1K1/111	P.c4–c5+	81.7143	white
pk1/1pP/111/P1P/1K1/111	k.b6–a5	−84	black
pk1/P11/111/111/1p/11K	k.b6–c5	0.442424	black
pk1/P11/11p/11P/11K/111	k.b6–c5	73.5484	black
pk1/P11/11P/1KP/111/111	k.b6–c5	0.748571	black
pp1/11k/11P/11P/P11/11K	K.c1–b2	72.5714	white
pp1/11k/11P/11P/PK1/111	p.b6–b5	−74.8571	black
ppk/111/11P/11P/P11/11K	k.c6–c5	−70.2857	black

89330 rows in set (0.07 sec)

(a) Q table

	2	471	64.4	−2.74	−64.5	2.99	88977	white
	2	472	−60	−2.74	63.33	3	88984	black
	2	473	−60	−2.75	61.38	3.02	88999	black
	2	474	−6.02	−2.76	−6.02	3.02	89018	NULL
	2	475	−0.62	−2.76	0.63	3.02	89091	NULL
	2	476	0.62	−2.77	0.64	3.02	89094	NULL
	2	477	61.5	−2.75	−61.46	3.01	89105	white
	2	478	62.44	−2.75	−62.25	3.01	89105	white
	2	479	61.62	−2.75	−61.58	3	89108	white
	2	480	61.91	−2.74	−61.9	3	89113	white
	2	481	0.61	−2.74	−0.67	2.99	89128	black
	2	482	0.65	−2.74	−0.65	2.99	89138	NULL
	2	483	−60	−2.75	61.74	3.01	89148	black
	2	484	−60	−2.76	62.67	3.02	89151	black
	2	485	60.95	−2.76	−60.9	3.01	89155	white
	2	486	−60	−2.77	63.64	3.02	89163	black
	2	487	0.67	−2.77	0.64	3.02	89172	NULL
	2	488	61.5	−2.76	−62	3.01	89179	white
	2	489	0.63	−2.76	0.62	3.01	89232	NULL
	2	490	−60	−2.76	61.9	3.02	89238	black
	2	491	0.62	−2.77	0.63	3.02	89278	NULL
	2	492	0.65	−2.77	0.63	3.02	89286	NULL
	2	493	61.62	−2.76	−61.58	3.01	89295	white
	2	494	60.95	−2.75	−60.9	3	89300	white
	2	495	62.44	−2.75	−62.25	3	89301	white
	2	496	61	−2.75	−61.17	3	89303	white
	2	497	0.64	−2.75	0.63	3	89314	NULL
	2	498	−60	−2.75	65.71	3	89314	black
	2	499	63	−2.75	−63.17	3	89314	white
	2	500	0.62	−2.75	0.64	3	89330	NULL

5500 rows in set (0.01 sec)

(b) P table

Figure 11: datatables in complex scenario

And then I inspect the tables generated which where much larger than the previous experiences: as shown in fig(12) R table reached five hundred thousands entries and Q table was just below 90000 entries.

After that, in fig(11b) there is the outcome of the five hundred greedy games. In which is clear that the training has given birth to a good policy, even if the difference between games won isn't remarkable.

```
| 507164 | 111/111/1k1/1p1/1K1/111 | K.b2-b1  |      1 | white |    500 |         2 |
| 507165 | 111/111/1k1/1p1/111/1K1 | p.b3-b2  |      1 | black |    500 |         2 |
| 507166 | 111/111/1k1/111/1p1/1K1 | K.b1-a2  |      1 | white |    500 |         2 |
| 507167 | 111/111/1k1/111/Kp1/111 | k.b4-a5  |      1 | black |    500 |         2 |
| 507168 | 111/k11/111/111/Kp1/111 | K.a2-b1  |      1 | white |    500 |         2 |
| 507169 | 111/k11/111/111/1p1/1K1 | k.a5-a6  |      1 | black |    500 |         2 |
| 507170 | k11/111/111/111/1p1/1K1 | k.a6-b6  |      1 | white |    500 |         2 |
| 507171 | 1k1/111/111/111/1p1/1K1 | k.b6-a5  |      1 | black |    500 |         2 |
| 507172 | 111/k11/111/111/1p1/1K1 | K.b1xb2  |      1 | white |    500 |         2 |
+--------+-------------------------+----------+--------+-------+--------+-----------+
507172 rows in set (0.46 sec)
```

Figure 12: R table

4 Conclusions

In the final analysis, the model seems to work very well for extremely simple scenarios, but since the state space size increments exponentially with the number of possible actions at disposal of the agent, with more complex experiments also the amount of training computation needed(and consequently training time) also grows exponentially.

The solution concerning the memorisation of data structures should be revised because the introduction of a Database connection causes an unacceptable overhead in terms of time. In fact, observing the computational resources used during learning, turns out that most of the time the python calls to database will wait for long time the response from the DB (time increases linearly with the size of the tables).

This would result in a poor management of the computational capabilities of the cpu which won't go over 30% of the maximum load and obviously won't even work when the thread is waiting for the DB response.

On the other hand, with tables stored in secondary memory the learning could be stopped and restarted at will. And, most important, the dynamic table could be accessed from different instances of the DB connection at the same time.

This fact hides an implication, there would be the possibility to carry on a distributed learning from different machines that will share the tables in the central Database, this means that every agent at time t will know what all the other agents have learnt until that instant speeding up the convergence.