Introduction

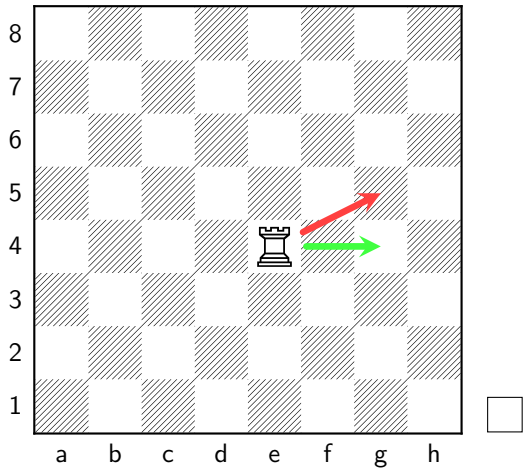
The goal of this GitHub project is to check if it is possible for a computer to learn chess rules, or in other words, if it is possible to make it played legal piece moves i.e. **without cheating**. The goal of this project is not to make predict moves or learn chess strategy to a computer (chessboard evaluation). This is already made by several projects. It is also interesting to compare how learning is expansive (in term of energy, iterations) in comparison to a human.
The goal of this document is to explain algorithms developed in the code source and to formalize reflections we had for this project. The axis of reflection of this document starts from simplified cases with modified chess rules and follows with more and more difficult cases.

1 Notation of a chess movement

A chessboard is a 8×8 grid and therefore has 64 squares. A chess movement is pair of squares of the board origin and destination that we will note for this document (o, d) with o and d positive integers < 64 . For example the green move of the rook on the next picture is ♖**e4–g4** with e4 is the 32^{th} square (starting from the 0^{th} square: a8) and g4 is the 34^{th} square. Therefore $(o, d) = (32, 34)$.

2 Teach moving a single piece on an empty board

Learning to move a single piece on an empty chess board with machine learning is the most simple case for teaching chess rules to a computer. We start with a neural network architecture.
We place randomly the desired piece on the empty chessboard. The computer tries a random move (legal or illegal) and a supervisor validates or invalidates the move which make reinforcing synapses: weights of synapses of the network are decreased or maintained. The supervisor knows the chess rules (while for this case slightly modified for accepting the non-presence of kings on the board).



The picture shows examples of random moves for a white rook: – in red the illegal move refused by the supervisor; – in green the accepted move.

Algorithm

Let A the matrix holding weights of the synapses, let e the input of the neural network and q its output. A holds all combination of movements therefore A is a 64×64 matrix. e is a vector of 64 elements (squares of the chessboard): we place the value 1 to the square holding the piece (therefore the origin o of the movement) and 0 when empty. q is the output: it will contain the normalized probabilities of the destination squares for finishing the movement.

The algorithm 1 for training a piece to move is the follow.

Algorithm 1: Train Synapses

$A_{i,j} = 1$ where $i, j \in \{0 \dots 63\}$
for X *iterations* **do**
  $o \leftarrow$ random a square
  $d \leftarrow \text{SynapsesPlay}(A, o)$
  $A_{o,d} = \begin{cases} A_{o,d} + 1, & \text{if } (o,d) \text{ is a legal move} \\ \max(1, A_{o,d}) - 1, & \text{else } (o,d) \text{ is an illegal move} \end{cases}$
end for

All synapses $A_{i,j}$ are set with a weight of 1. We random the origin o of the movement. From o and A we random for the destination of the move. We decimate synapses when the obtained move is an illegal chess move. We reinforce synapses when a legal move is found. The sub algorithm 2 gets the destination of the move.

Algorithm 2: Synapses Play

Input: A: synapses, o: origin of the move
Output: d: destination of the move
Initialization:
$e_i = \begin{cases} 1, & \text{if } i = o \\ 0, & \text{else} \end{cases}$ where $i \in \{0 \dots 63\}$
Matrix production with normalization of the result:
$q = A \times e$
$S = \sum_{i=0}^{63} q_i$
$q_i \leftarrow q_i / S, \forall i \in \{0 \dots 63\}$
Randomize the destination move:
$p \leftarrow \text{random}(0,1)$
$q = 0$
for $(d = 0;\ d < 64;\ d = d + 1)$ **do**
  $q = q + q_d$
  if $q \geq p$ **then**
    return d
  end if
end for
return *no move*

From o, we initialise the input vector e by placing 1 to the o^{th} element and 0 others. We compute probabilities q for the destination of the move by doing the matrix product of A and e. From normalized q, we random a the destination of the move.
The following figure shows an extract of the matrix A for a Rook after 10^5 iterations.

	A8	B8	C8	D8	E8	F8	G8	H8	A7	B7	C7	...	H1
A8	0	8	60	128	69	47	100	313	77	1	1		
B8	6	0	2	263	75	111	138	156	0	75	0		
C8	54	3	0	111	139	15	48	96	1	0	183		
D8	132	254	107	0	124	9	236	21	0	1	1		
E8	73	78	156	130	0	88	28	17	0	0	0		
F8	56	115	18	10	89	0	68	142	0	1	0		
G8	105	150	52	244	35	81	0	21	0	1	0		
...													
H1													

We read this matrix as follow: the column is the origin of the move and the row the destination. For example a Rook on the a8 square cannot go to a8 (0) but can go B8 or C8. Nevertheless, we can see that the matrix still have initial values 1 (for example A8-B7) meaning that this branch has not been explored and potentially can produce an illegal move.

Code source

The code of the supervisor is made in src/Chess/Rules.cpp and the code of the training is made in src/Players/NeuNeu.cpp. Enable the macro RANDOM_MOVES.

Optimized algorithm

With this simple algorithm that we need to iterate over millions of iterations to obtained a correct weights for synapses A and explore all cases. If not, a branch of movements may not have been tried and not decimated in the case of erroneous movement. As consequence the computer may choose an illegal movement. This is of course not acceptable because considered as cheating by a human player.
We could improve the algorithm of the gradien but a simpler idea comes immediately: why not simply iterate on all 64×64 cases and eliminate definitively synapses when the move is illegal ? This solution works nicely and only takes 4096 iterations.

```
    A8  B8  C8  D8  E8  F8  G8  H8  A7  B7  C7 ... H1
A8   0   1   1   1   1   1   1   1   1   0   0 ...
B8   1   0   1   1   1   1   1   1   0   1   0
C8   1   1   0   1   1   1   1   1   0   0   1
D8   1   1   1   0   1   1   1   1   0   0   0
E8   1   1   1   1   0   1   1   1   0   0   0
F8   1   1   1   1   1   0   1   1   0   0   0
G8   1   1   1   1   1   1   0   1   0   0   0
...
H1
```

Real game

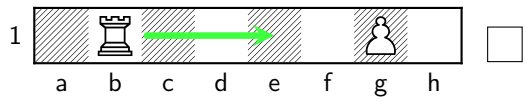
This algorithm can be applied on all pieces: Rook, Knigth, Bishop, Queen and King. Training a single color is sufficent for playing the oponent. Note that the color of the square for Bishops is agnostic. For Pawns we have to create a network by type of color because the direction are oposite.
Once trained, the machine can almost play against a human. We random a piece along the available pieces. This makes the origin of the move. We apply the algorithm 2 SynapsesPlay for the destination. Because we did not teach not to move a piece to move to a piece of the same color, the supervisor, knowing chess rules, makes iter again a new move.

Code source

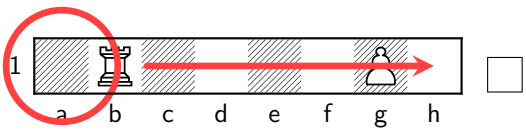
The code of the supervisor is made in src/Chess/Rules.cpp and the code of the training is made in src/Players/NeuNeu.cpp. Disable the macro RANDOM_MOVES.

3 Teach to move a single piece on an empty board

Now, the difficulty is to teach how to move a piece with obstacles. By obstacle we mean the presence of another piece of the same color.
Again, to start simple, we simplify the problem to a 1-Dimension problem: the chessboard is no longer a 8×8 board but a 8×1 board. And to simplify again the problem the piece only can move to the right. The two next pictures show idea.



On this picture, the green arrow shows a random and valid move.



On this picture, the green arrow shows a random and invalid move.