Introduction

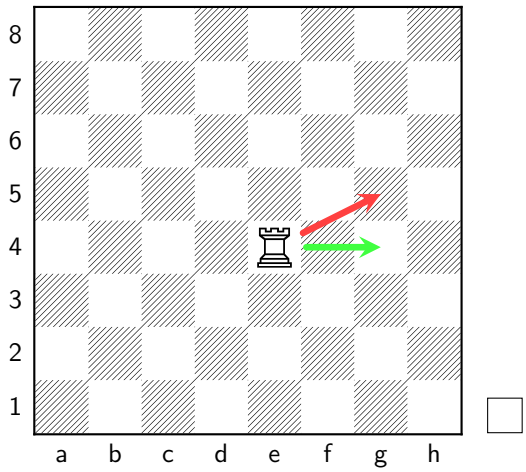
The goal of this document is to formalize and explain algorithms developed in the code source of this GitHub project developed in the aim to know if it is possible for a computer to learn completely chess rules by itself, or in other words, if it is possible to make it played legal chess movements *without cheating*. Its goal is not to make move predictions or to make learn chess strategy to a computer (chessboard evaluation), this has already been done. This document allows to reflect upon how learning is expansive (in term of energy, iterations) for a computer in comparison to a human.

1 Notation of a chess movement

A chessboard is a 8×8 grid and therefore has 64 squares. A chess movement is a pair of chessboard squares: the origin and the destination. For this document, and contrary to the official chess notation, we will encode the movement as follow: (o, d) a pair of strictly positive integers and lower or equal to 63. We count squares left to right, top to bottom (i.e. from 0^{th}, the a8 square, to 63^{th}, the h1 square). For example the green move of the Rook on the next picture is classically written **♖e4–g4** but for this document, the square e4 is the 32^{th} square and g4 is the 34^{th} and therefore this move is encoded $(o, d) = (32, 34)$. The reason is to be usable in mathematics as coordinates for 64×64 matrices.

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 simplest case for teaching chess rules to a computer. For this case, no external library is needed: a home-made fully connected neural network is enough.
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.

2.1 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 possible movements therefore A is a 64×64 matrix. e is a vector of 64 elements (squares of the chessboard): we place the value 1 at the index associated to the origin square o of the piece movement to move and 0 in others. q is the neural network output: it will contain the normalized probabilities of the destination squares for finishing the movement: from 0 for coding higly improbable move (forbidden) to 1 for higly probable (mandatory move).

The algorithm 1 for training a given 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

Let us explain algorithm 1. 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 SynapsesPlay(A, o) algorithm given in 2 allows gets the destination d of the move. The move is validated (legal) or invalidated (illegal) by an supervizor (an hard-coded function knowing the chess rules).

Algorithm 2: Synapses Play

Input: A: synapses, o: origin of the move
Output: d: destination of the move
`/* Initialization */`
$$e_i = \begin{cases} 1, & \text{if } i = o \\ 0, & \text{else} \end{cases} \text{ where } i \in \{0 \dots 63\}$$
`/* 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 */`
$p \leftarrow$ 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*

Let us explain algorithm 2. Knowing the origin o of the move, we initialise the input vector e by placing 1 to the index refering to o and 0 for other elements. We compute probabilities q of movement destinations by doing the matrix product of A and e and normalizing the result q. From probabilties of the destination d of the move (o, d) of select randomly a move.
The following figure shows a extract of the Rook synapses (the 64×64 matrix A) 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. Let suppose a Rook placed on the a8 square. Illegal moves like ♖**a8–a8** is correctly understood: the weight is 0. Legal moves such as ♖**a8–b8** or ♖**a8–c8** are correctly understood: weights are > 0. 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 could produce an illegal move.

2.2 Code source

- the supervisor src/Chess/Rules.cpp
- the training is made in src/Players/NeuNeu.cpp. You have to enable the macro RANDOM_MOVES.

2.3 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. 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 gradient descent 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. Here synapses for the Rook:

```
    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
```

2.4 Real game

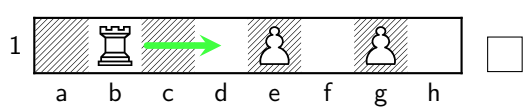
This algorithm can be applied for all type of pieces. Each piece owns its own synapses. For Rook, Knight, Bishop, Queen and King the training for a single color is sufficient: it is the same for the opposite color. Note that casttle moves, promotion and takes are not learnt. For Pawns, we have to create a synapses depending on the color because they have opposite move directions. Note that for Bishops, the color of the square where is placed the piece is agnostic.
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 on/over a piece of the same color, the supervisor, knowing chess rules, has to force iterating a new move until a legal one is found.

2.5 Code source

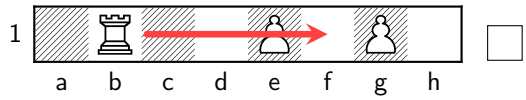
- the supervisor src/Chess/Rules.cpp
- the training is made in src/Players/NeuNeu.cpp. You have to disable the macro RANDOM_MOVES.

3 Teach to move a single piece on an obstructed board

The next step is to teach how to move a piece with obstacles. By *obstacles* we mean the presence of another pieces of the same color. Again, to start with the simplest case, we focus with a 8×1 chessboard (therefore to a 1-Dimension problem) and we suppose that the moving piece is a Rook-like piece which can only go to the right. Chess rules concerning obstacles is maintained (the Rook cannot cross the piece of its color). The two next pictures summarizes the principle.



On this picture, the green arrow shows a valid move. The a1 square is not explored for this problem (right-only movement).



On this picture, the red arrow shows an invalid move because in chess rule a Rook, contrary to the Knight, cannot jump over other pieces.

3.1 Convolutional Neural Network

Contrary to the previous section, where a fully connected neural network have been used, we are going to use a convolutional neural network (CNN). CNN have done a great improvement for machine learning concerning image recognition: they allow to create numerous small kernel filters detecting small portion of images (such as edges). These filters have the property to be position invariant inside the image. CCN are so powerful that nowadays many non grid machine learning problems are translated into a grid problem. Fortunately for us, a chessboard is already a 2-Dimension grid. The following document is a very well good introduction to CNN, their convolution and their pooling functions.
In our case, the CNN is made of the following layers:

- The input of CNN is made of two layers representing each of them a 8×1 chessboard.
  - The first chessboard/layer contains the blocking pieces. We do not care of the type of the piece bu only care of their presence or absence on each square of the chessboard: we encode 1 for their absence and 0 for their presence. For example the layer of the previous figure will be $\begin{bmatrix}1 & 1 & 1 & 1 & 0 & 1 & 0 & 1\end{bmatrix}$
  - The second chessboard/layer contains the moving piece and we encode 1 on the index of its presence and 0 for other squares. For example the layer of the previous figure will be $\begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$
- The first layer is a convolution operator with a window of 8 squares and a padding of 7 because the chessboard is not infinite and we have to take care of borders. Therefore we extend the layer dimension to 8+7 squares for the Convolution operator.
- The second layer is a max pool operator.
- In practice CNN ends with a fully connected layer neural network as final layer. For simplicity reasons, this will not be our case: we only compute the entropy loss function.
- the output of CNN returns the probability of the number of squares for the movement from 0 (meaning a move of 0 square in the case the piece cannot move) to 7 (meaning no blocking piece and the moving piece is on the a1 square). For example the layer of the previous figure will be:

  $\begin{bmatrix}0.01 & 0.01 & 0.9 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01\end{bmatrix}$

3.2 Evolution – WIP

This CNN finds the distance to the first blocking piece. For knowing the possible squares to move we have prefered to have the following result: $\begin{bmatrix}0.01 & 0.45 & 0.45 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01\end{bmatrix}$ because the Rook can move to squares c1 or d1 (1 square and 2 sqaures).

3.3 Training and testing

We do not need a database of chessboards for training and testing this CNN. The problem is so simple that we can generate by ourselves chessboards and the expected answer needed by the supervisor. For the first layer, we simply generate random numbers 0 or 1. For the second layer is initially zeroed and we place a single 1 randomly.
The supervisor function simply compute the distance (the number of squares set to 0) between the Rook and first blocking piece. For example the layer of the previous figure will be $\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$ encoding the fact the distance is 2.

3.4 Newbie error concerning the training

For this problem it is important to generate a board with several blocking pieces. If we generate only a single blocking piece, the CNN will learn **a** distance between the moving piece to the blocking piece but not **the** distance between the moving piece to the **first** blocking piece. As consequence, if we train the CNN with a single blocking piece and test it on a obstructed board, the CNN will not be able to detect the first blocking

piece and all distances will have the same probability to appear and therefore the CNN will not be able to move correctly the piece.
While the case of a single blocking piece, the CNN will converge faster and explore more possibilities than an obstructed board, this is a totally different problem. Indeed, the probability of having a chessboard with 1 or 0 blocking pieces is very few contrary to a chessboard where the piece cannot move. As consequence, we have to be sure to generate a sufficient number of training where this case is produced for the training. For example with 20 thousands of iterations we can obtain 40 cases of empty chessboard. As consequence a bad choice of iterations can produce unexplored case.
As consequence, we see that we have two choices: – either have an exhaustive set of data where edges case appears the same occurance than usual case – or either over millions of iterations. We point out the limit of machine learning.

3.5 Test humans

What we succeeded to make understand to the machine is the algorithm of the distance to first blocking piece placed on the right. This simple algorithm needed around 5 lines of Julia or C code.
It can be interesting to test humans with the same method: do not explain the goal of the game but simply show the figure and ask him a number between 0 and 7. Wait his answer and simply say correct or incorrect. Count the number of iterations until he understand the algorithm. Of course, the chessboard has to be shown differently. Firstly, human does not have to see this problem as a chessboard with Rook and Pawns because he will immediatly understand the problem if he already knows how to play chess. Instead just replace the figure of pieces by dots and crosses to make the problem totaly abstract.
Not tested, but probably, a human, contrary to the computer will not need thousand of iterations before understanding the solution because a human brain is good for localizing lines in a pattern. A more difficult variant of this problem could be the distance to the second blocking piece. The human brain is less good for this job and probabaly (not tested) a human will not succeed this test.

3.6 Code source

The code of this section has been made with Julia 1.0 and the Knet package. See ChessNeuNeu/scripts/ChessKnet.jl. This program have not yet been translated in C++.

4 To be continued