Introduction

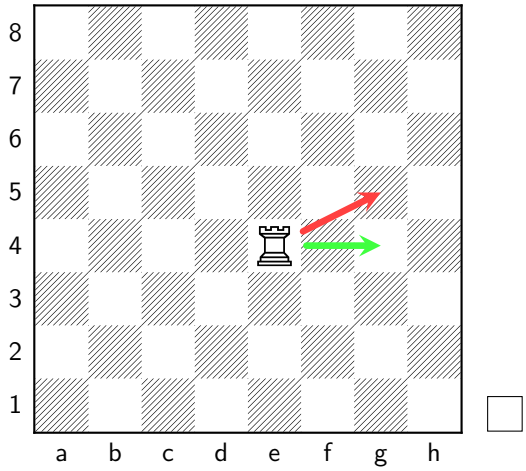
The goal of this document is to formalize reflections and explain algorithms developed in the code source of this GitHub project. The goal of this project is to check if it is possible for a computer to learn chess rules, or in other words, if it is possible to make it played legal piece moves **without cheating**. The goal of this project is not to make predict moves or learn chess strategy to a computer (chessboard evaluation) because this has already been made. It is also interesting to compare how learning is expansive (in term of energy, iterations) in comparison to a human.

1 Notation of a chess movement

A chessboard is a 8×8 grid and therefore has 64 squares. For this document we define a chess movement has a pair of squares inside the board: origin and destination. We will note it as (o, d) where o and d are positive integers and < 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 use a fully connected 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.

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 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:
$p \leftarrow$ 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*

Knowing 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 by randomizing a number between 0 and 1.
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. 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 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.

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

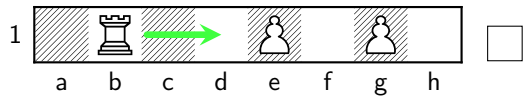
This algorithm can be applied for all type of pieces: Rook, Knight, Bishop, Queen and King. Training a single color is sufficient for playing the opponent. For Pawns we have to create a network by type of color because directions are opposite. Note that the color of the square for Bishops is agnostic.
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 to move to a piece of the same color, the supervisor, knowing chess rules, makes random a new move.

2.5 Code source

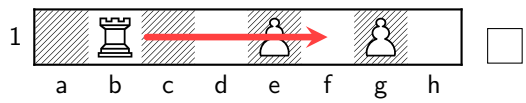
The code of the supervisor is made in src/Chess/Rules.cpp and the code of the training is made in src/Players/NeuNeu.cpp. Disable the macro RANDOM_MOVES.

3 Teach to move a single piece on an obstructed board

The next step is to teach how to move a piece with obstacles. By obstacle we mean the presence of another pieces of the same color. Again, to start with the simplest case, we focus with a 1×8 chessboard (therefore to a 1-Dimension problem) and we suppose that the moving piece can only go to the right. Chess rules concerning obstacles is maintained. The two next pictures show the principle.



On this picture, the green arrow shows a valid move. The a1 square is not explored for this problem.



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 and their convolution and pooling functions.
Our CNN is made of the following layers:

- Input: two 1×8 chessboards. The first chessboard contains the blocking figures. We do not care of the type of the piece (Pawns, ...). Only their presence on squres is important: 1 and their absence is 0 for each square. The second input contains the moving piece. Layer is in fact we increase tensor of 7 elements (passing from 8 to 15) because of the next layer: conv.
- First layer: Conv. Padding is 7 because the chessboard is not infinite and we have to take care of borders.
- Second layer: max pool.
- Thrird layer: fully connected layer.
- Output: returns the probabily of the number of squares for the movement from 0 (meaning a move of 0 square. Or in the case the piece cannot move) to

3.2 Training

We do not a database of chess games for training this CNN. The problem is so simple that we can generate chessboards and their solution for the supervisor.
x = zeros(8,1,2,1) x[:,1,1,1] = Int.(floor.(0.5 .+ rand(8)))
p = Int.(floor.(1 + 8 * rand(1)[1])) x[p,1,2,1] = 1
The supervisor calcy training function will return y
For example in the previous figures y will be [0010000]
Say differently, this CNN learns the distance to the first blocking piece.

3.3 Testing

3.4 Newbie error for 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 the 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 training with a single blocking piece and testing on a obstructed board, the CNN will not be able to detect the first blocking piece and will not be able to move correctly the piece.
While one blocking piece the CNN will converge faster and explore more possibilities than an obstructed board, this is a different problem.

3.5 Drawback of this method

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. A good choice of data for training shall include edges cases.

3.6 Test human

convolutional
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.
We can compare in how many iterations are needed for a human to understand the
It can be interesting to test humans with the same. Of course, the problem has to be shown differently. Firstly, human does not have to see it is a chessboard. So replace the chessboard of figure XX by a simple grid uniformly white and replace the piece to be moved (Rook) by a circle and blocking pieces (Pawns) by a cross. Show this

figure to a human and simply ask to him “give me a number between 0 and 7”. Repeat again until he understand that the answer is the number of empty squares to the first blocking pieces. Count the number of iterations needed.
A human 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.

3.7 Code source

The code of this section has been made with Julia (version ¿= 1.0) and the Knet package. See ChessNeuNeu/scripts/ChessKnet.jl