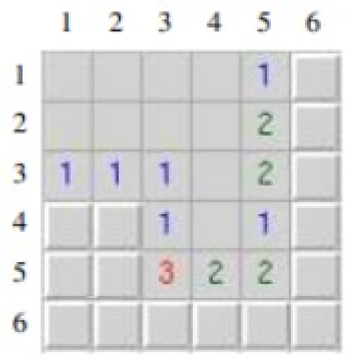
Minesweeper solvers

In this repository you will find two solvers for the mineweeper game. Explanations about this game on its official page: *https://minesweeper.online*. I will present here the approach and the results I obtained with two different methods, constraint and satisfaction and reinforcement learning solvers.

Constraint and Satisfaction problem (CSP)

The CSP solver uses logic and constraint satisfaction problems. This method could be more like the thinking a human might do when playing a game of minesweeper.

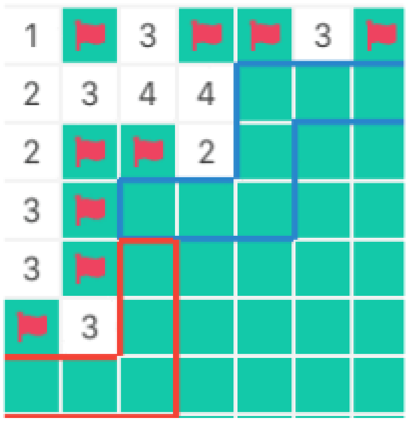
Firstly, this model will try to discover all the cells that are certain to be a mine or not. This is done through simple equation solving. Let's take the example below:



$$\begin{array}{ll}
(1,5) & x_{16} + x_{26} = 1 \\
(2,5) & x_{16} + x_{26} + x_{36} = 2 \\
(3,1) & x_{41} + x_{42} = 1 \\
(3,2) & x_{41} + x_{42} = 1 \\
(3,3) & x_{42} = 1
\end{array}$$

Here, an equation can be determined for each square adjacent to a discovered square. Indeed, from the square **(1, 5)**, we can determine that only one mine is located in the squares **(1,6)** and **(2,6)**, hence the first equation. By applying this principle to all our squares and solving our system of equations we can already find the position of many mines.

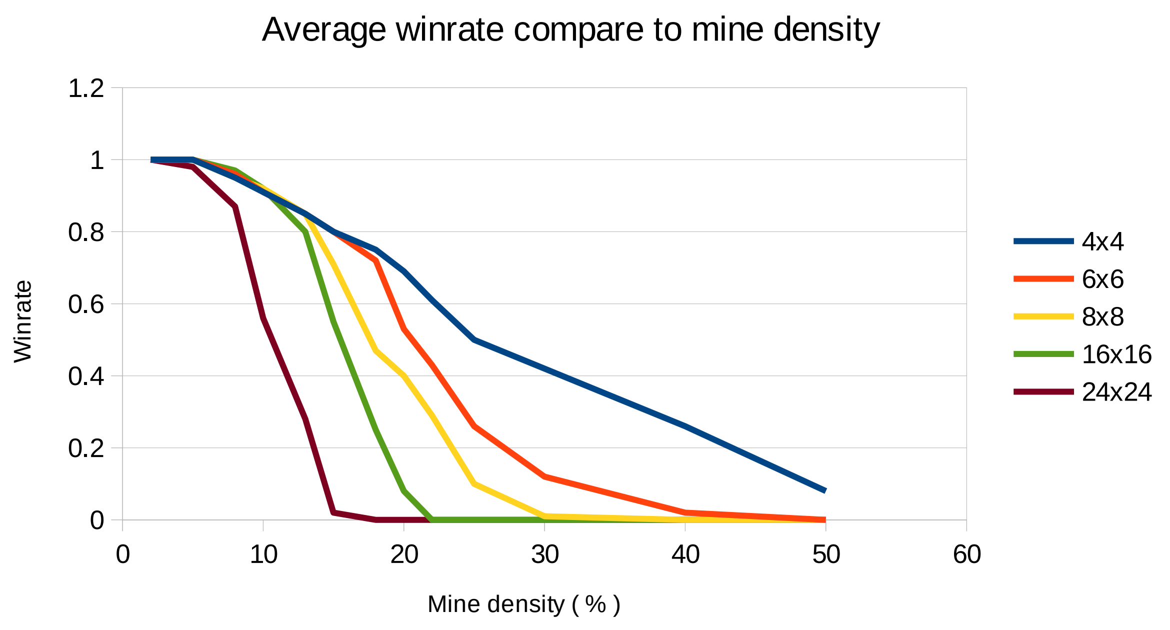
When no more squares are found, the model will then calculate the probability of the presence or not of a mine in each square. To calculate this, we look at all the combinations of mine positions that could satisfy the resolution of our game. This works very well but is very resource intensive material. Indeed, let's imagine that there are, for example, 32 unknown squares adjacent to a discovered square. This brings us to a total of *4 294 967 296* possibilities. To counter this, the grid has been divided into independent parts, see example below:



From this example we reduce our group of 12 squares (i.e. 4096 possibilities) into two groups of 7 and 5 squares (i.e. 182 + 32 = 214 possibilities). From all these possibilities, the model can deduce the probability of the presence of a mine on each of the squares, and choose the lowest one to play the next move.

The paper *“Playing the Minesweeper with Constraint”* [1] allowed me to understand how constraint programming techniques can be applied to the game of minesweeper. I followed this paper’s ideas (especially parts 3.1, 3.2 and 3.3) to implement this solver.

Results



Double Deep Q-Learning

In our case, the limitation of reinforcement learning would be the size of our Q-Table. Indeed, for very small grid sizes (3x3 for example) this would be possible. But for larger sizes the number of transitions that our Q-Table would have to store would be much too large.

To get around this problem the goal here is to train a **neural network to predict our Q-Value**. This removes our Q-Table and the limitation say above. The implementation of such a neural network is called Deep Q-Learning.

However, two new limits are now emerging: **overfitting** and **overestimation**.

Indeed, if our neural network trains itself to predict our Q-Value, and this for each of the actions one after the other, the strong correlation between the transitions of the same part then poses problems of overfitting. To avoid this, we add a buffer which will store a certain number of transitions and which will train our network on a random part of this network at each move. We are now in Deep Q-Learning with experience replay.

In our case we use the same neural network to evaluate our Q-Value in *t* and *t+1*. However, this is a problem if the prediction of our future Q-Value (in *t+1*) is too big this will fit our network to predict a Q-Value in *t* that is also overestimated. By a kind of chain reaction the next time our Q-Value in *t* will then serve as the future Q-Value in *t+1*, and once again this will lead to an overestimated prediction. To limit the influence of our network on itself, I used a second network. The second network allows us to take care of estimating only the future Q-Value. Periodically the weights of our main network will then be copied into this second network in order to continue the good evolution of our training. Here, we reach our final model, a **Double** **Deep** Q-Learning model with **experience replay**.

To help me understand the thought process around the resolution of such a game with reinforcement learning, this article referenced in [2] helped me a lot. You will find in this article everything I said above in different words and more precisely explained.

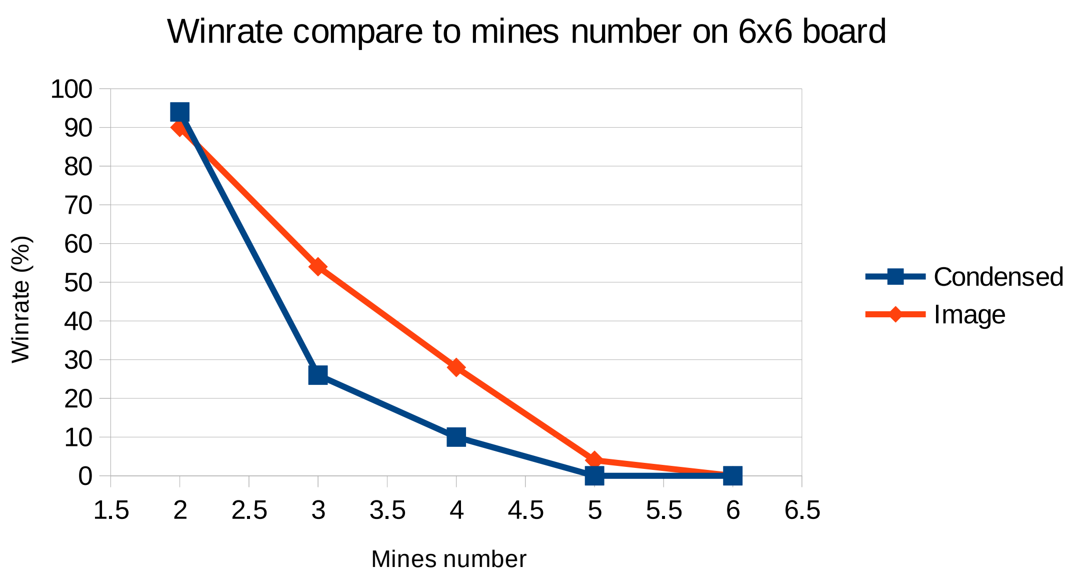
The neural network

I built my first network following the observations made in the article [2]. The grid can be treated as an image so we use a network of four convolutional layers 128 filters, 3x3, and no max-pooling (to keep the information intact) followed by two dense layers of 512 hidden units and an output layer of size of our grid. The minesweeper board is return to our neural network as an *N x M x 1* matrix with integers 1 to 8 for known field and -1 for unknown field. To help the convergence I decided to normalize this data between 0 and 1 for known field. Here the use of four convolutional layers in a row could lead to a vanishing gradient problem. To help our network in performing a correct back propagation of the gradient, residual neural network has been implemented. The concatenation of the first and second layer serves as input to the third layer as well as the third and fourth layer serves as input to the first dense layer.

The second network topology was taken form the paper *“Evolution strategies and reinforcement learning for a minesweeper agent”* [3]. The grid can still be treated as an image so we use here a network of two convolutional layers of 18 and 36 filters, 5x5 and no max-pooling followed by three dense layers of 228, 220 and 220 hidden units with an output layer of size of our grid. The minesweeper board here is return to our neural network as an *N x M x 2* matrix. First channel for the integers, and the second channel has 1 if field is unknown and 0 otherwise.

Results

I trained these two models on 6x6 grids with different mine density and here are the results :



Shaping

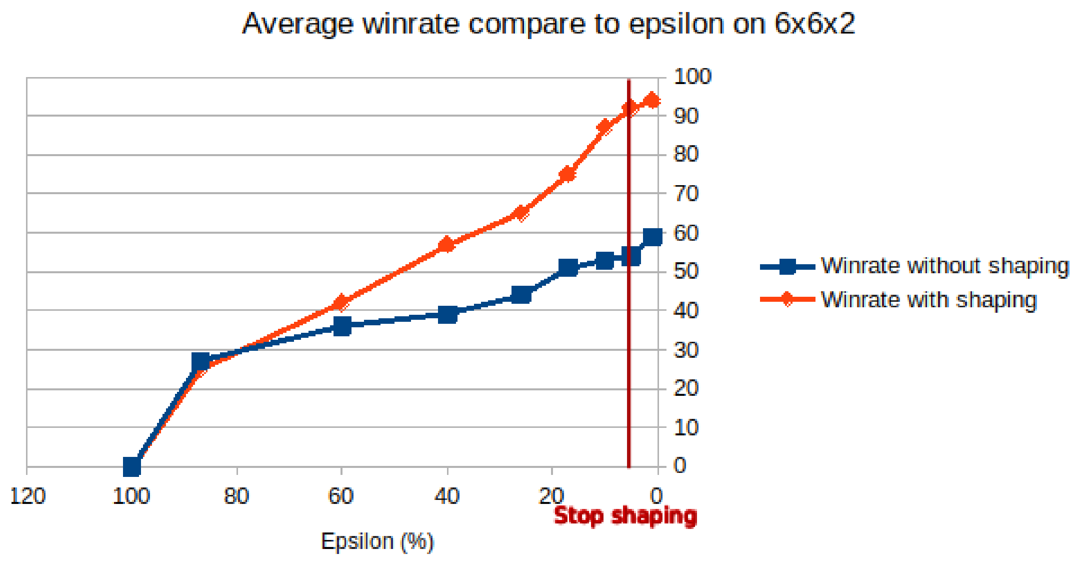
The performance of the CSP solver for medium board size and density are correct unlike our deep reinforcement learning model whose performance collapses as the size or density of mines increases. To help this poor model I tried to adapt my reward structure as followed :

'csp': 1.2, *'win': 1, 'lose': -1, 'progress': 0.9,' guess': -0.4, 'no_progress': -0.3*

The aim was to help our model in its exploration phase. As the CSP can makes perfect choices, greater rewards would then be given to our agent if it made the same choices. When the exploration phase comes to an end (epsilon < 5%), we return to the starting structure.

'win': 1, 'lose': -1, 'progress': 0.9,' guess': -0.4, 'no_progress': -0.3

Here are the results :

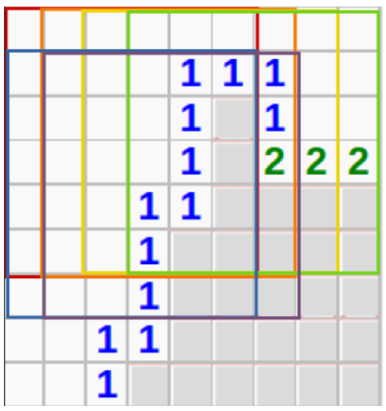


Here it can be seen that when the exploration phase is guided, by the shaping, the winrate increase faster. At the end, the results obtained are more than 30% better than the results without shaping.

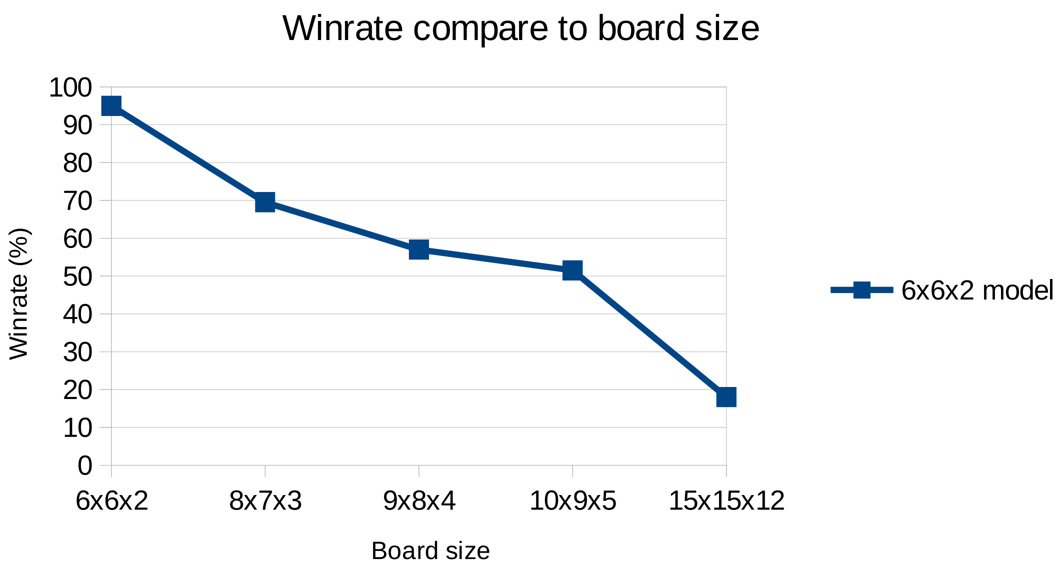
Transfert learning

Currently the solver is only train on grids of size 6x6. In order to solve larger grids two solutions have been tested.

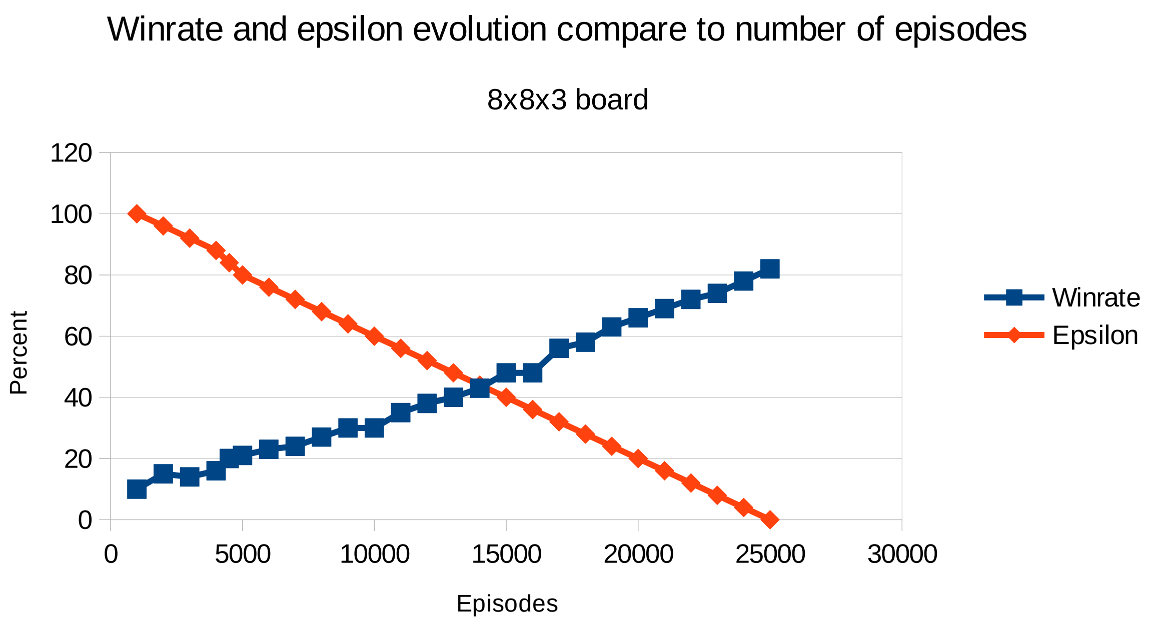
First one is not present on this repository because the results were too satisfactory. The grid has been cut in part of 6x6. For example, for an 8x8 grid, it is split into 9 sub-grids of 6x6 (see figure below) and predictions are made on each cell of each sub-grid. A cell can therefore have up to 9 predictions. In this case, an average of the predictions is made for each cell and the best average is chosen in order to perform the next action.



See below the results of this method on different boards with the same mines density:



However, if you wish, you can always do a little work to change the input and output numbers of the network so that it matches your minesweeper grid. For example, I trained a model to play on 8x8x3 board and here are the results.



Work in progress

- Use of a technique for rapidly transferring the information stored in one neural network into another neural network.

- Experiment others networks topology, rewards structures and hyper parameters.

- Implement a Bayesian solver and try to do shaping with this solver.

References

[1] https://www.info.ucl.ac.be/~pvr/minesweeper.pdf
[2] https://sdlee94.github.io/Minesweeper-AI-Reinforcement-Learning/
[3] https://github.com/jakejhansen/minesweeper_solver