Module 6 report

Håkon Gimse og Kari Skjold

We have written this assignement using python 3.5 and Theano.

Artificial neural network design

In module 5, we experienced that rectify and softmax as activation functions worked very well. We found that this configuration worked best in this module as well. The reason why we chose these activation function is explained further in the report for module 5. We experimented with various learning rates. In module 5, we found that learning rate 0.01 gave the best results. This learning rate was therefore a natural starting point when starting working on module 6. The error function used was sum of squared error. This makes sure the network does not learn to fast, and overfits to the training data. We also experimented with the number of hidden layers and the number of nodes in each network. We found that a single hidden layer performed best with ca 700 nodes, but anything above 100 gave more or less the same results.

Generate training cases based on situations the neural networks experiences

In our first solution, we let the AI from project 4 play games, and log the game configuration, and the heuristic values from each move to a file. After a while we figured that a problem was that the expectimax algorithm was a lot smarter than the neural net. The result of this was that the ANN often got in a situation the Expectimax had never gotten in. The ANN did not now what to do, as it had not trained on these situations. The solution was to let the training cases be generated by using the board configuration the ANN got into, and the result from the expectimax to that configuration.

General

We have explored with various training data. First, we tried using our data from expectimax where the heuristic promotes a snake-look-alike board. However, this pattern turned out to be a little too hard to understand for our neural network. Therefore, we wanted to try having a very simple heuristic as a starting point and then work from there on. We decided on a gradient-based dataset, because the board configuration and the chosen move seamed to have a higher coorelation than with the snake. It will always try pushing tiles against the top left corner, while the snake wants every other row to be decreasing/increasing. A very small change in the configuration can change the direction. The ANN is unlikely to detect these changes and predict the wrong move.

Representation one

In the first representation we just wanted to pass the tile values and positions through the network. In order to do this, we created a network with 16 input nodes, each representating a position on the board and the value of this tile.

This performed reasonably well, but an issue was that as the largest tile grew larger, all the smaller tiles was nearly ignored in the network. To normalize this to the range [0, 1] we reduced every value to the logarithm of the value and divided by the larges tile, like so:
$f(x) = log_2(value)/log_2(max(board))$
This representation achieved a largest tile average of ca 200.

Representation two
An unwise move the neural network often did with the last representation is that even when it is able to merge a lot of tiles to improve the game, it sometimes prioritize the clear gradient structure too much. That is, a clear diagonal structure which is shown in the table below.

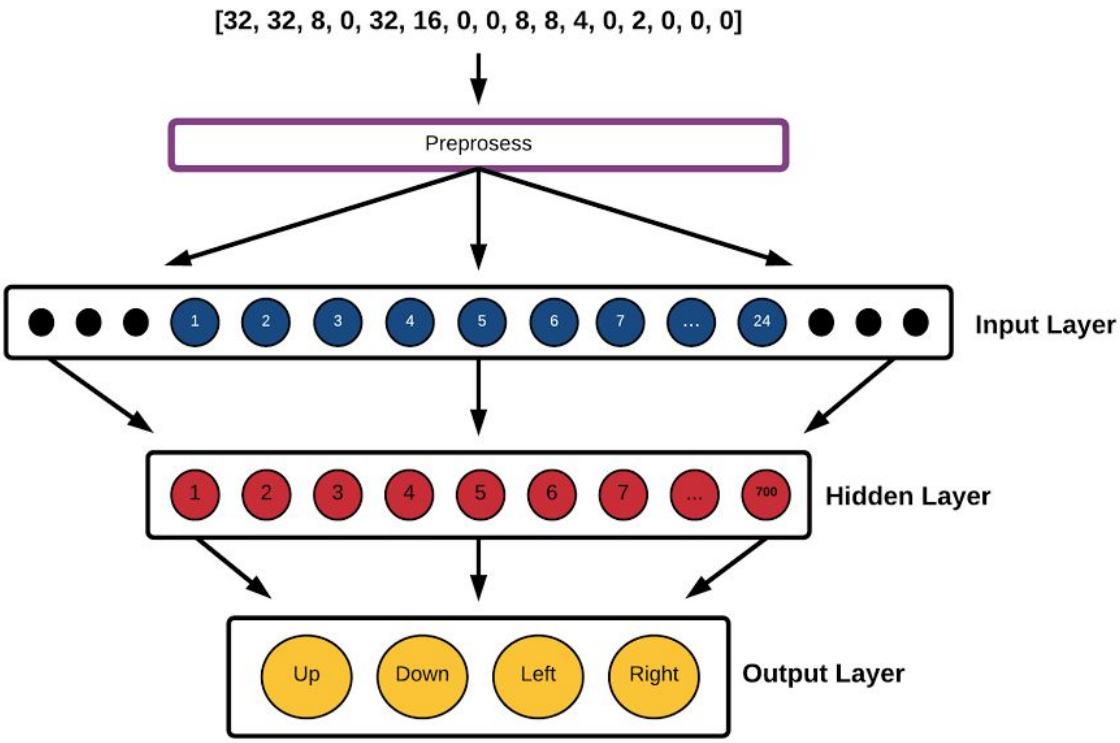
256	128	64	32
128	64	32	16
64	32	16	32
32	16	32	8

Consequently, it is not able to merge and the board is very easily filled up which results in game over. To solve this problem, we decided on creating a new training set where the tiles in the upper row had some extra bonus points in the heuristic. This heuristic results in a more skewed gradient so that the board wants higher tiles on top (it favours the up move), but still wants to merge the tiles in a gradient-ish direction towards the top left corner. This new training set improved our neural network a lot. We now achieve an average highest tile of 450.

To make it clearer for the neural network that particular moves leads to great merges, we created eight more input nodes, one node per column/row telling the neural network how good merge score it gets by doing the move.

The calculated merge score in a row/column is the sum of the logarithmical value of the tile-pairs that can be merged in the row/column. For example, the row [64,64,2,2] will create an input node with value log2(64) + log2(2) = 7.
This value is also scaled to the range [0, 1] by dividing by the highest tile on the board times two, as this is the maximum merge score a row/column can have. We think that with more nodes, ergo more information for the neural network, the pattern will be even clearer for the neural network. The eight extra nodes are also very consistent with the training data because it only strengthen the positive effects of merging towards the top left corner.

This representation is a great improvement from representation 1. We achieved a largest tile average of ca 450. Graph 2 in the attachments shows a graph of the comparison between representation 1, representation 2 and a random player.



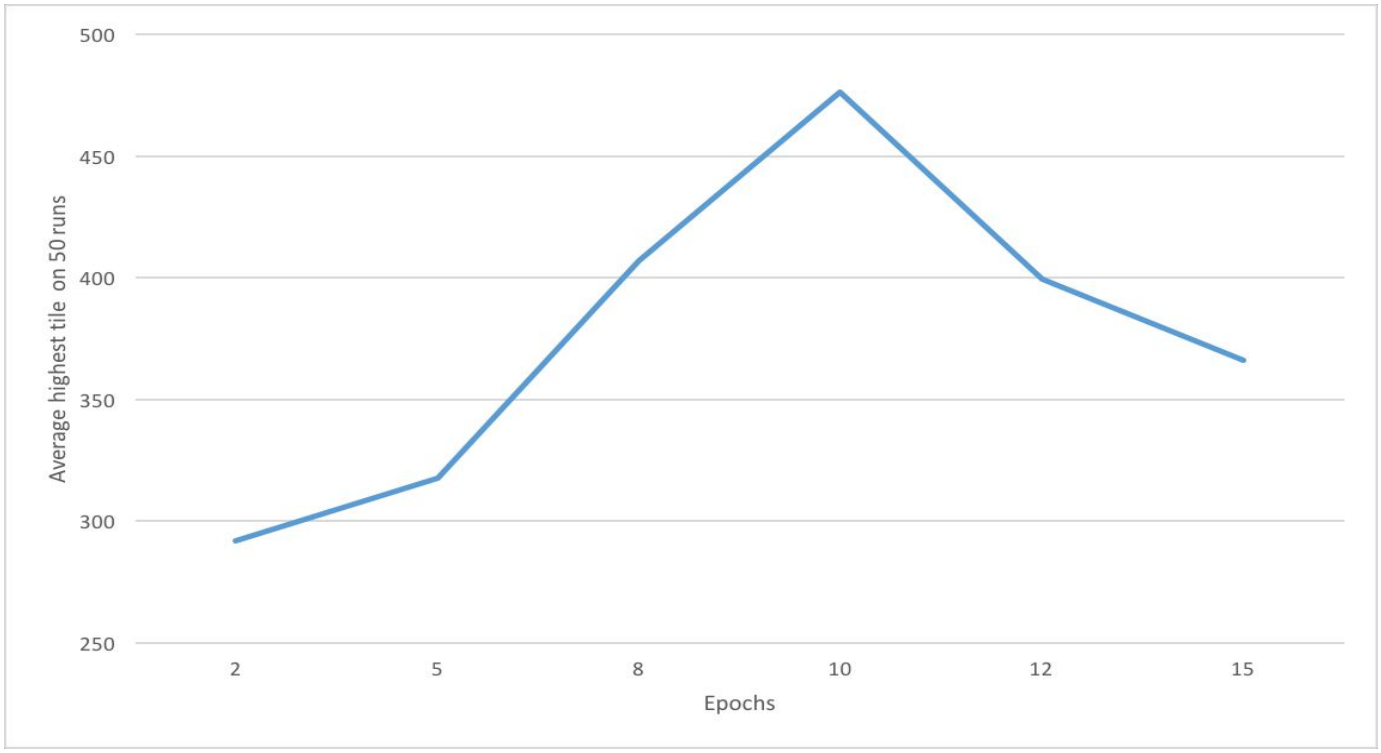
Smart decisions

The neural network realises very quickly that it should keep the highest tile in the top left corner. This is because we have a very strict gradient (towards the top left corner) heuristic when generating the training data with expectimax. It never moves right if the top row contains empty tiles and it never moves down if the left column contains empty tiles because then it ruins the position of the highest tile.

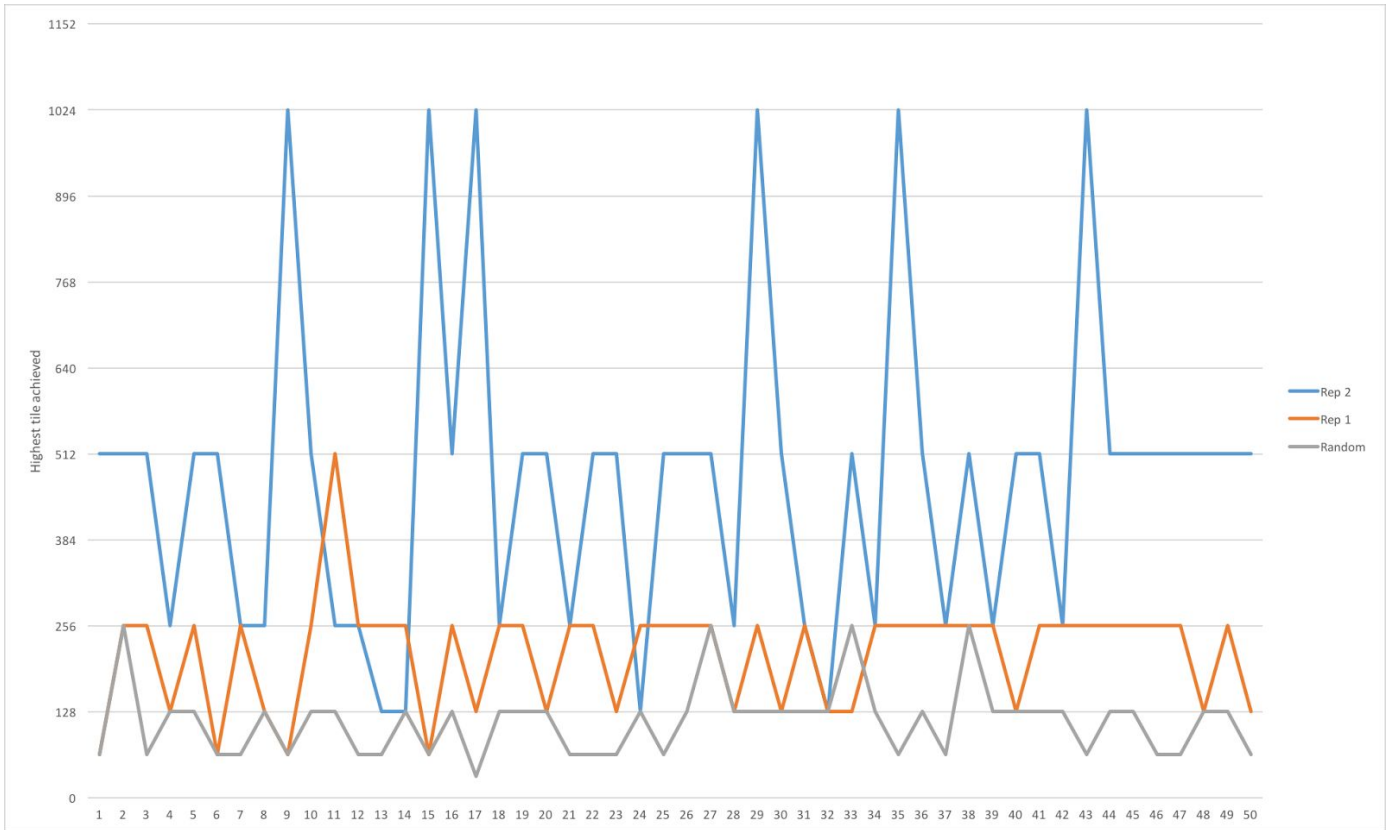
Unwise moves

A problem that often occurs, is that small tiles are locked in by higher tiles. When this happens, the neural network is not able to fix it because it will always keep pushing tiles towards the top left corner. The ANN almost never try to “fix” bad states. It is very consistent and straight forward, and makes the same move everytime.

Attachments:



Graph 1: We found that to train our neural network 10 epochs gave the best result.



Graph 2: Comparison between representation 1(orange) and representation 2(blue) and random(gray).