Module 6 - Deep Learning for Playing 2048

Aleksander Skraastad
Håkon Ødegård Løvdal

November 23, 2015

1 Introduction

The purpouse of this report is to describe our work done with module 6 in IT3105. In this report we will elaborate about our design choices, game state representation and an analysis of the game performance of the ANN.

2 Design choices

When designing our network, we primarily used our findings and experience from module5 as a starting point. Additionally, several resources online stated that for high abstraction features such as handwriting, voice regocnition etc, 2 hidden layers with node count between 1.5 and 3 times the input layer have proven very successful. An additional design choice was limiting the number of hidden layers (for training speed), while still maintaining the network's ability to "remember" the finer details of the training data. Since we had great results with the accelerated backprop from module5 (RMS prop), we opted to keep it as is, without further modifications. Learning rate when using this method plays a much smaller role, as it is dynamically scaled based on the degree of error.

With regard to activation functions, we found that for deep networks, the rectified linear unit activation function performed very well, in addition to being extremely fast.[1]

Building on that foundation, the rest of the network design was very much trial and error. Understanding what aspects of performance were caused by topology, AF or representation was a lecture in routine and patience.

When we were approaching our final data representation, we settled on a base design of 64-192-128-4 (3x relu, 1x softmax), with 192 and 128 being hidden layers, which gave the best results. This in turn corresponded nicely with the 1.5x to 3x the input node count for hidden layers that we used as an initial design foundation.

[1] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

3 Data selection

Due to the fact that the AI created in module 4 proved to be better in the game of 2048 than either of us, we chose to use that game data.

In essence, we wanted the network to adjust its weights to ideally correspond to the "Gradient" weight matrix used to generate the training data. Initially, with raw board representation, results were poor. But we could see the network was getting into trouble for not choosing one side over the other. We adjusted the weight matrix with a bias to one side, and generated about 600 000 board states.

This proved to be very effective, and when watching the network play we could clearly see that it still used a gradient approach, but stacking tiles towards the right-side edge of the board.

4 Representation

At first, we attempted representing the board as a 16-element array containing the numerical values of the board. Several different topologies, activation functions and parameters were tested. Initially, this gave poor results, with correctness converging at 50%. We understood that the amount of nodes was not enough to properly "encode" the correct gradient weights. Ramping up to a 192-128-64 (hidden) network gave decent scores, with tile average of 347 in 1000 runs.

We started out replacing actual values with a mergeability index for each cell using the formula:

$$m(x,y) = \sum_{i=0}^{4} \frac{min(cell(x,y), neighbor_i(x,y))}{max(cell(x,y), neighbor_i(x,y))}$$

More research led us to expanding the input vector from 16 to 64 nodes. One 16 element slot was reserved for the mergeability index, using the above formula. The other three remaining 16 element slots were filled with board values, the "Gradient" weight matrix and free-cell counts and number of merges possible. This gave even better results, yielding just above 60% and average tile value of 488.

A final breakthrough came when properly scaling the input values using the natural logarithm. We then opted to reduce the input size to 48, and simply provide the network with the results of a move up, move right and move down, only presenting the network with how the game state would look in three of the four directions. Without any heuristics. The network was then gradually able to learn the gradient heuristic.

$$\text{Weight} = \begin{bmatrix} 8 & 16 & 32 & 128 \\ 4 & 8 & 24 & 64 \\ 2 & 4 & 16 & 32 \\ 1 & 2 & 8 & 16 \end{bmatrix}$$

Table 1: "Gradient" weight matrix

Table 1 illustrates the weight matrix used to generate the training data. Our goal was for our network to learn to replicate this weight matrix.

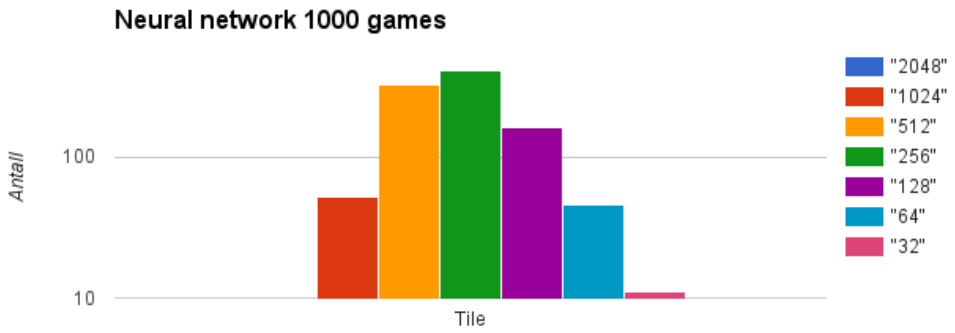


Figure 1: ANN (64-192-128-4), 30ep, 56%, 600k samples - Plain representation

Figure 1 shows the tile distribution when using the original 16-element numeric values representation with avg. highest tile of 347 in 1000 games.

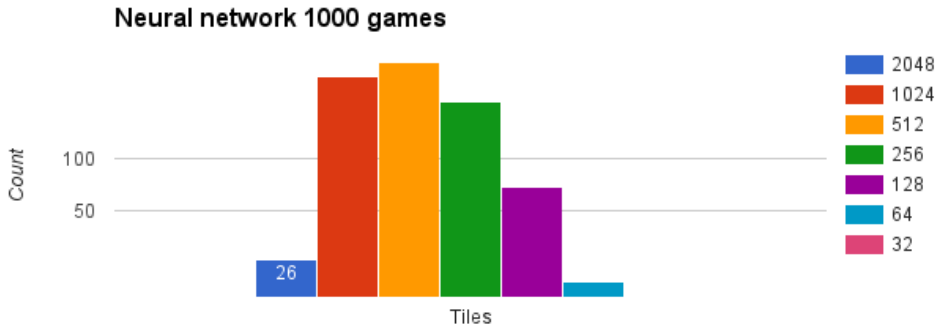


Figure 2: ANN (64-192-128-4), 30ep, 68%, 600k samples - Advanced representation

Figure 2 shows the tile counts after 1000 games with the final representation. The network was trained for 30 epochs, with a learning rate of 0.00001 (Using RMS propagation, rho=0.9,

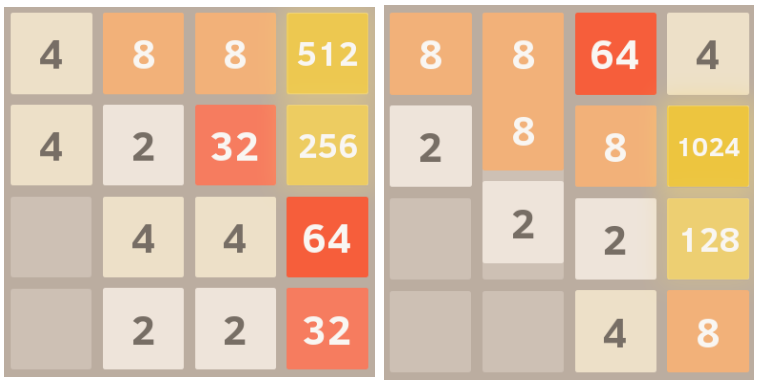
epsilon=1e-6), in about 10 minutes. 2048 was achieved a whopping 26 times, which gives a 2048 tile percentage of 2.6%, thus clearly beating the 64 tile percentage of 1.9%. Avg. tile was 616.38.

Comparing the original representation to the new and improved representation, we had an easy choice in selecting what representation to use.

We also tested to add the gradient weight matrix as an additional 16-element bias vector. But this gave insane results of above 800 avg. tile on 1000 runs. Fearing point deduction on the report (in case this was helping too much), we chose not to include it.

5 Game analysis

The trained network plays using a gradient strategy with a bias towards the right side relatively well. It is in most cases able to stack tiles in the top right corner, and arrange lower tier tiles in descending order on the right hand side. It does however sometimes run into trouble, for example if it performs a down move, or left move, and either gets a tile spawn in top right corner or mistakeably positions a low tile there. This is demonstrating the fact that the network has not learned the gradient weight matrix properly.



(a) Good tile placement (b) Bad tile placement

Figure 3: Examples of network tile placement

Figure 3a shows how the network is able to correctly stack tiles in descending order, with a bias to the right hand side of the board.

On lower value tiles, the network is quite often able to get itself out of trouble, by means of the gradient effect. It continues to stack tiles in the top of column 3, eventually building up enough to be able to merge. But for higher tile value such as the one in figure 3b, it simply does not stand a chance.