**Maksymilian Mika, Martyna Mikoda, Stanisław Wilczyński**

# Project report - Neural Networks and Deep Learning 2017 at University of Wrocław

QLearning for the “Threes!” game

[Introduction](#)

[First approach](#)

[QLearning](#)

[Network architecture, meta parameters and data preprocessing](#)

[Results](#)
- [Learning from scratch on generate playouts](#)
- [Learning from scratch on minmax playouts](#)
- [Learning from minmax in minmax playouts](#)

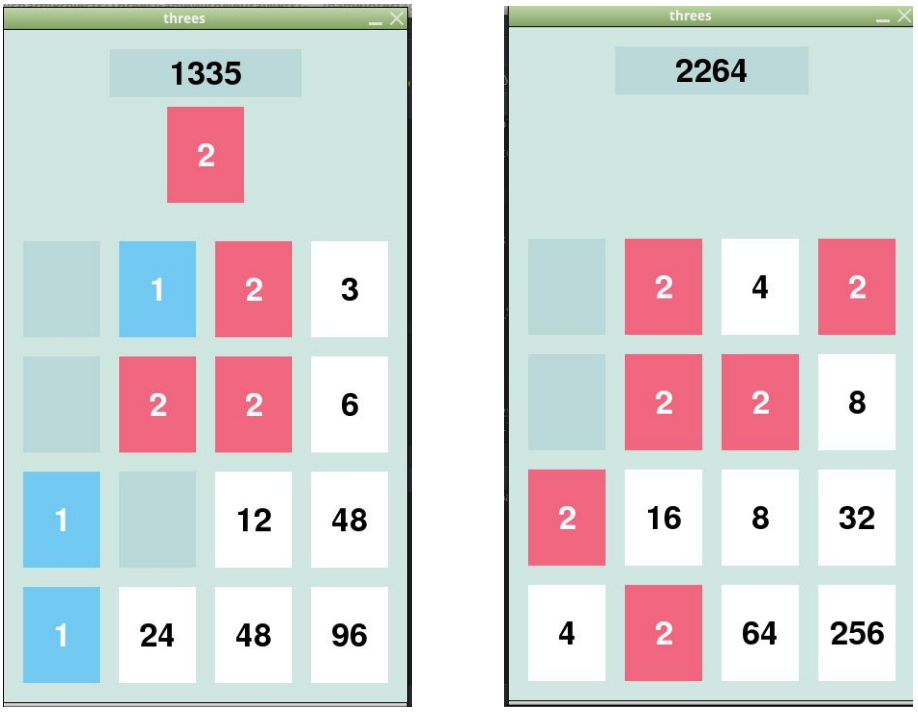
[Summary](#)

[Acknowledgments](#)

## Introduction

In our project we implemented an engine for the “Threes!” game and “2048”. We used algorithm described in ([1]) to train our model based on simple neural network to play these games and compared its effectiveness with random agent and minmax agent. The whole code can be found in our repository.

We created a general model and graphic interface for all games similar to Threes and 2048.



Game interface for Threes and 2048

# First approach

In order to have some comparison we first tried to solve the game with regular, non machine-learning methods. We tried simple MinMax algorithm with heuristic function that takes into account not only the value on the board, but also whether the large values are near the edges. With depth 4, the results were comparable to those achievable by very skilled players.

| | RandomAI | SimpleAI | MiniMaxAI | NeuralNet |
|---|---|---|---|---|
| Score | 929 | 2276 | 15154 | 3155 |
| Max Value | 89 | 179 | 499 | 240 |

# QLearning

The goal of the QLearning algorithm is to learn a function $Q : (S, m) \rightarrow R$ which returns a value that estimates how many points till the end of the game can be obtained if we make a move $m$ from state $S$. In order to do so, we created a simple 2-fully-connected-layer neural network with ReLU activation function.

We used the following learning criterion to train out network based on generated playouts. For every transition $(s_t, a_t, r_t, s_{t+1})$ where $s_t$ is the state from which we performed move $m_t$ and got instant reward (points for merged blocks) $r_t$ and got to state $s_{t+1}$ .Let $y_t = r_t$

if $s_{t+1}$ is a terminating state. Otherwise $y_j = r_t + \gamma \cdot \max_{a'} Q(s_{t+1}, a')$ . Then the loss can be expressed as $L = (y_t - Q(s_t, a_t))^2$ . The process of network training starts with initialization of the weights of our layers to small random values. Then until requested number batches was processed (in one batch we put 100 transitions), we feed the network with one batch and we use ADAM optimizer to perform gradient descent steps on our parameters. Our playouts (training data) were generated using $\varepsilon$-greedy algorithm - we create a new game and choose moves according to current Q-values, and with probability $\varepsilon$ (in our case $0.1$) we choose a random move instead. To reduce the correlation between transitions within one batch, we used $Replay\ Memory$ technique from ([1]). At every iteration we were replacing randomly chosen subset of remembered transitions with newly generated one. In order to reduce the computational burden, our network is fed with states only and generates for outputs (one value for every move). This allows to reduce the number of forward passes through the network during training.

# Network architecture, meta parameters and data preprocessing

As our network used only state (represented by float values) as the input, the layers of the network had sizes $19 \times 256$ (or $16 \times 256$ in case of “2048” network) and $256 \times 4$ respectively. The additional three in case of “Threes!” stand for information about next blocks that may appear on the board. As for the metaparameters we tested different values of learning rate and weight decay to make the results of training more reliable. We also used two different version of normalization (squeezing our inputs to $[0, 1]$). In the first one the score (integer value) was divided by maximum score in the game, and number (integer value) in each block was divided by maximum possible number in block. In the second approach we followed the normalization from ([2]); we took the logarithm of each number in the block.

# Learning

## Learning from scratch on generated playouts

The first way of learning the net was to initialize it with random data and then by playing try to teach it the optimal QValue for the game. The data in this approach was generated by the net. To generate next batch of transitions the net played the game choosing moves that produced best QValue results. Additionally the $\varepsilon$-greedy strategy was employed. ([1] ) With probability equal to $\varepsilon$ we chose random move instead of the one dictated by the QValue. This way the nets QValues were used every time the game generated the next batch for learning. So the $Replay\ Memory$ was filled with the plays generated by some portion of recent nets. We then tried to learned the net for various learning rates.

## Learning from scratch on minmax playouts

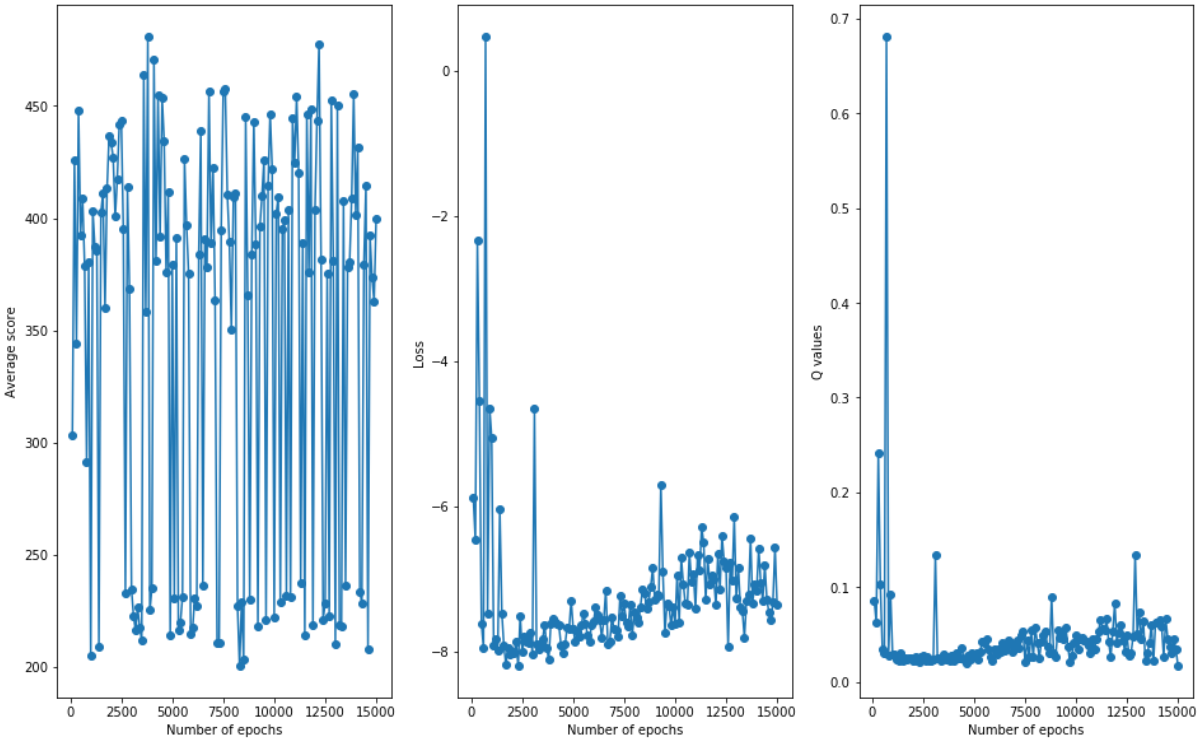
Since results from the previous method were not satisfying we employed a second way of learning the net where the data in $Replay\ Memory$ was filled with the transitions taken from an algorithm that played with MinMax strategy.

Unfortunately, this process took too long to properly train the network.

## Learning from minmax in minmax playouts

In all previous approaches the net was updating the parameters of the network according to its own predictions of Q. ($y_j = r_t + \gamma \cdot \max_{a'} Q(s_{t+1}, a')$) We also tried replacing the bit with our own estimation of Q which was calculated by a MinMax algorithm. To estimate value of $Q(s_t, a)$ the algorithm started in state $s_t$ and then took action $a$. Afterwards it continued playing with some simple heuristic and when the final board was met the result was set as $Reward - Result(s_t)$, where $Reward$ is the result of the final board, and $Result(s_t)$ is the result of $s_t$. (The result for a game state for both Threes! and 2048 is well defined.)

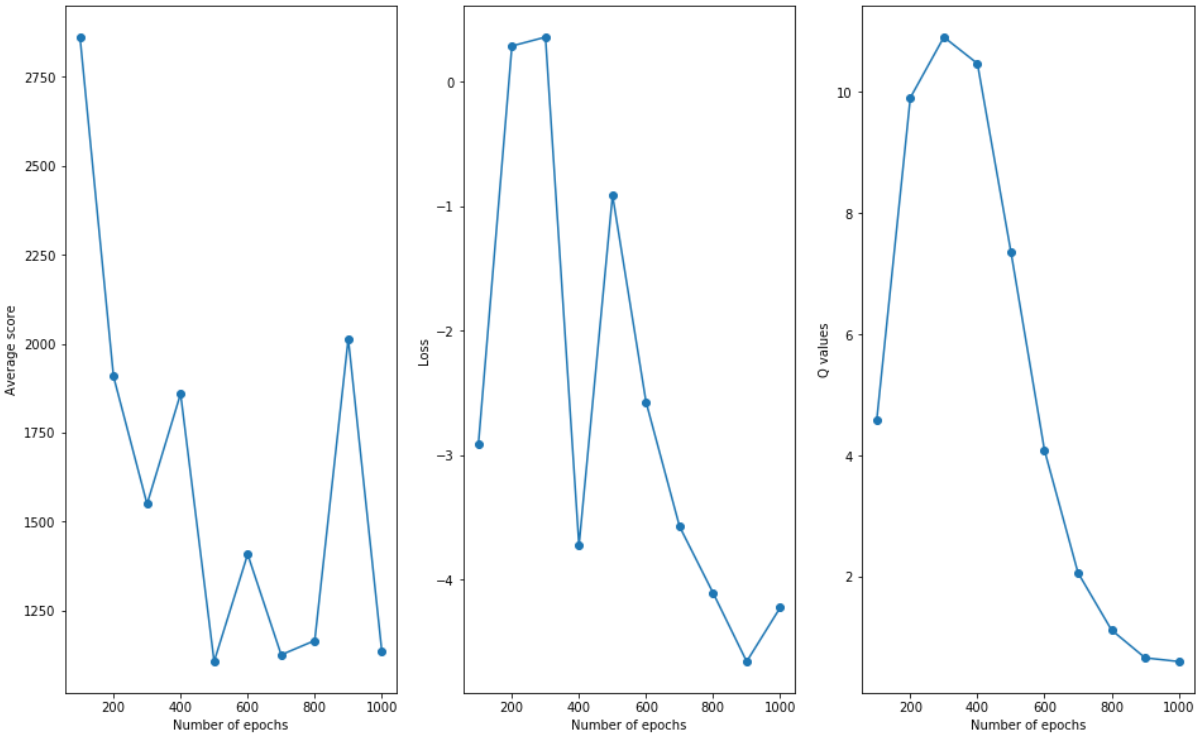
Unfortunately, this process took too long to properly train the network.




Results before repairs for Threes!

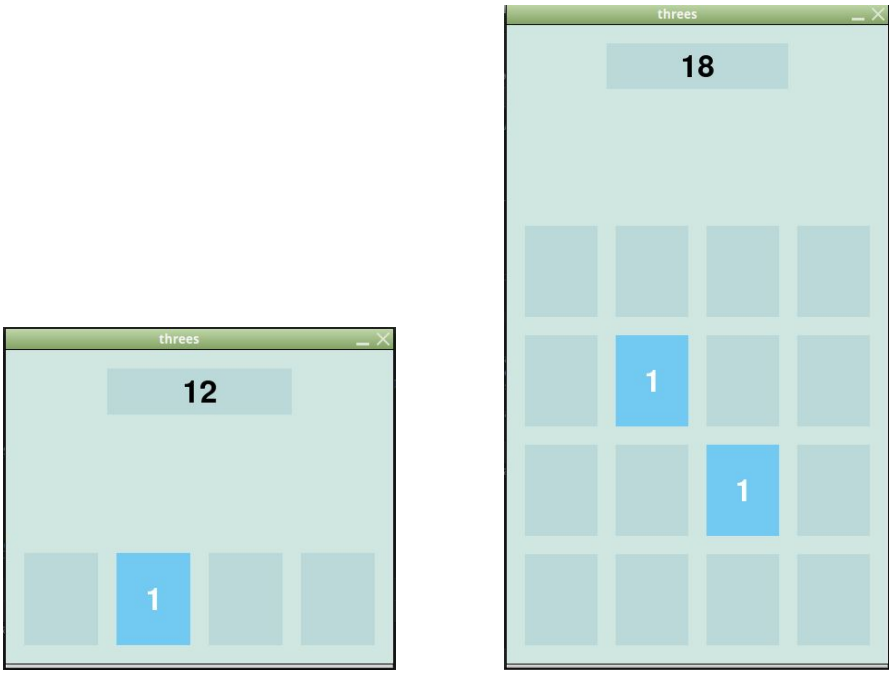
# Test

Since the results weren't satisfying, we tried to train a network on easier games. It helped us identify many bugs; unfortunately, we didn't have a chance to fully train our fixed network.




Initial results for fixed version for 2048

We created two easy games. Goal of the first one was to move piece to the right corner. The network learned it pretty easily. In the second one there were two blocks, and user had to join them. The network learned to move blocks to the right corner instead of the closest one.




Easy games

# Summary

Neural nets are better than Simple AI figured out by a person. They cannot compete with Minmax algorithm as for now. Further improvements are necessary, because the method is promising.

# Acknowledgments

The authors thank Google for GCE Credits awarded through Google Cloud Platform Education Grants to the Neural Networks and Deep Learning course and to this project.

# Bibliography

1. Playing Atari with Deep Reinforcement Learning V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. (2013)cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013
2. Georg Wiese's github