2048 Report

Karl Allik
Raul-Martin Rebane
Robert Sepp
Lembit Valgma
Supervisor: Ardi Tampuu

January 11, 2018

1 Introduction

Researchers at Google DeepMind have lately made significant advances in computer played Go using deep reinforcement learning [11, 12]. Most recently with AlphaGo Zero [9], they developed an algorithm that starts with only knowing the rules of Go and by simulating the games, finally learns to play the game with superhuman skill. AlphaGo Zero beat the previous version of AlphaGo 100 to 0, while the previous version beat the human grand champion Lee Sedol. Since the algorithm starts only with the rules of Go and receives no extra human input, it could be viewed as universal solution for some set of games.

Our project has two main goals

1. Understand AlphaGo Zero solution algorithm.

2. Apply AlphaGo Zero algorithm (with minimal modifications) to game 2048 [7].

There are some differences between Go and 2048, as the latter is a single player game and contains some randomness. The differences might be large enough that the AlphaGo Zero algorithm does not generalize. However, 2048 has a much smaller state space than Go, which should allow us to use a much shallower neural network and get results without requiringa as much computational power.

2 Background

Most common deep learning applications have been based on supervised learning, where we use (human) expert knowledge (like a labeled data set) to train the algorithms. This usually gives good results but limits the applications a lot, since human expertise is expensive and in some places impossible to get. Reinforced learning (RL) systems are trained from their own experience, rewarding the outcomes that are closer to desired results. They have achieved great success where behavior of the systems can be simulated but is difficult to analyze (e.g. number of possible states is huge).

In RL framework we usually model our problems as Markov Decision Process (MDP). We have a set of states S and set of actions A. In the beginning of each time step the system is in some state $s \in S$. Then we take some action $a \in A$ after which the system ends up in state $s' \in S$. Upon each time step we can also earn some reward $R(s, a, s')$. The behavior of the system and our choice of actions can both be probabilistic. Probability of ending up in the state s' after taking action a in state s is denoted by $\Pr(s'|s, a)$, this is description of environment. Our rules for choosing the action a in state s are described by policy $\pi = \Pr(a|s)$.

General goal for solving MDP is that we want to come up with a policy that maximizes expected total cumulative reward. In the general case, the state space is too big to solve the problem directly and a common approach is to simulate MDP many times and choose the best policy based on the simulated results. It is known that the process converges, although sometimes extremely slowly.

2.1 Reinforcement learning in AlphaGo Zero

With some simplifications, the state space S for Go can be described by placement of black and white pieces on the board, each cell in a 19×19 grid can either contain black, white or no piece. Action space A can be described by a 19×19 grid, where each cell represents the probability of putting a piece on that cell. Reward is 1 when no more moves can be played and player has won, 0 on all other moments. Policy π is described as a deep neural network that takes $s \in S$ as input and outputs move probabilities $\Pr(a|s)$. In addition to move probabilities, AlphaGo Zero neural network also outputs player win probability v (expected rewards from state s).

The neural network is randomly initiated and trained from (simulated) self-play games. In self-play, from each state, to choose next move, Monte-Carlo Tree Search (MCTS) is executed. The search is guided by the current neural network and outputs probabilities for the next move. Our intuition is that MCTS gives us much better move probabilities (for that state) than neural network. Now we simulate the game until the end and on each state remember the results from MCTS. Once the game is over, we can use MCTS results and the winner to retrain the neural network. Actually we simulate many games and then retrain network. Finally, over iterations, the neural network should start to behave more and more like MCTS. For more specifics on NN and MCTS used in AlphaGo Zero, see [12, 8].

2.2 Adjustments for 2048

2.2.1 Rules of 2048

"2048 is played on a gray 4×4 grid, with numbered tiles that slide smoothly when a player moves them using the four arrow keys. Every turn, a new tile will randomly appear in an empty spot on the board with a value of either 2 or 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move.

The game is won when a tile with a value of 2048 appears on the board, hence the name of the game. After reaching the 2048 tile, players can continue to play (beyond the 2048 tile) to reach higher scores. When the player has no legal moves (there are no empty spaces and no adjacent tiles with the same value), the game ends."[1]

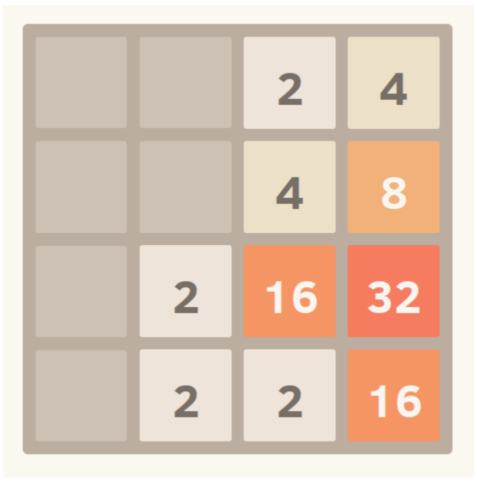

Figure 1: A game of 2048 in progress

2.2.2 Concerns

There are differences between Go and 2048 and this raises some questions that need to be solved.

1. Randomness of environment. New tiles appear randomly in each state. Need to carefully select neural network architecture that can generalize properly.

2. Simulation speed. AlphaGo Zero executed around 5 million self plays. Their implementation was highly efficient and parallel. It is difficult to estimate the requirements for 2048, as the state space is much smaller. But efficient simulator is still needed.

3. Selection of reward for MDP and loss function for neural network. The reward should be relatively linear and separate states well, thus highest tile is not the best choice. Score seems natural to use. However, since one neural network needs to output move probabilities and expected reward, the scale needs to be considered.

3 Methods

Since our goal was to understand AlphaGo Zero (AGZ) solution and try to apply similar approach for 2048, our first design was an adaptation of AGZ solution to 2048. The first part of this section describes our process in more detail. Unfortunately we did not manage to get satisfactory results from this solution. We still wanted to use neural networks to play 2048 and used slightly simpler process to train the network. The second part of this section describes that process.

3.1 Neural network with Monte-Carlo search tree

Our goal is to train a neural network that takes a 2048 game state s as input and outputs both move probabilities and a value, $(\boldsymbol{p}, v)$. The scalar value v represents the ”goodness” of state s. In AZG v represented the probability of winning from the state but we defined (since we have only one player) it as a ratio between the expected current game result and the average result using current network. This way, values above 1 are good and below 1 are not so good.

The network is trained from games of self play by a reinforcement learning algorithm. During self play in each state a Monte-Carlo tree search (MCTS) is executed which is guided by the current neural network. MCTS produces probabilities π for each move. These probabilities should theoretically be much stronger than the probabilities $\boldsymbol{p}$ that the network outputs. Thus, the simulator chooses next move based on MCTS results π and also saves the state s and probabilities π. Once the current game is simulated until end, the ratio of final score and average model score, z is added to saved states and probabilities, giving triples (s, π, z). These triples represent training data for network. Once some number of games are simulated until end, the simulation is stopped and network is retrained using a random batch of triples from recent games. Then the new network is evaluated against current one by playing a few games, if the result of the new network is better then current network is replaced and used to guide simulations in following iterations.

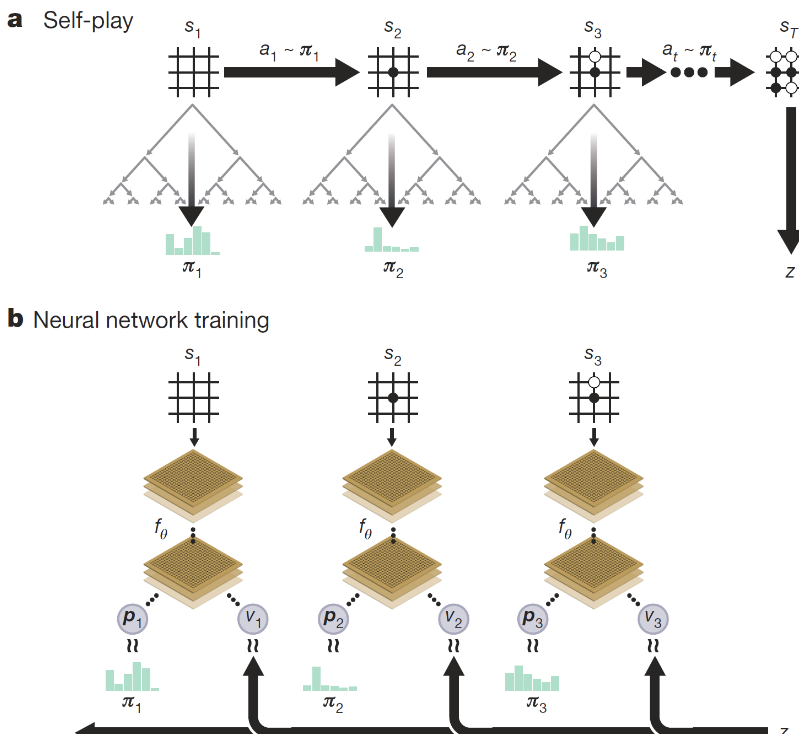

Figure 2: Self-play reinforcement learning pipeline [12]

3.1.1 Monte-Carlo tree search

In each state MCTS is executed. Basically a tree is built, the root node is the initial state s_0. For each state s in the tree, there are 4 edges (s, a) which correspond to possible moves. Each edge has statistics

- $N(s, a)$ - number of times action has been taken from state s,

- $W(s, a)$ - total value of move a from state s,
- $Q(s, a)$ - mean value of move a from state s,
- $P(s, a)$ - prior probability of selection move a.

Algorithm iterates over three phases (initially 1600 times)

1. **Select**
   Each iteration starts at s_0 and finishes when simulation reaches a leaf node s_L. At each timestep t move is selected according to the statistics in the search tree,
   $a_t = \text{argmax}(Q(s_t, a) + U(s_t, a))$ where
   $$U(s, a) = P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}.$$
   This search control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

2. **Expand and evaluate**
   The leaf node s_L is evaluated using the current neural network, yielding move probabilities p_a. Leaf node is expanded and each edge (s_L, a) is initialized to
   $N = 0, W = 0, Q = 0, P = p_a$. The value v is then backed up.

3. **Backup**
   Edge statistics are updated in a backward pass for each $t \leq L$
   - $N(s, a) = N(s, a) + 1$,
   - $W(s, a) = W(s, a) + v$,
   - $Q(s, a) = W(s, a)/N(s, a)$.

   Once the iterations finish, probabilities $\boldsymbol{\pi}$ are calculated
   $$\boldsymbol{\pi}_a = \frac{N(s_0, a)}{\sum_b N(s_0, b)}.$$

Next move is simply selected by maximum $\boldsymbol{\pi}_a$.

In training phase, AGZ uses sampling from the distribution $\boldsymbol{\pi}$. We used a deterministic selection since there is much randomness in 2048 already anyway.

3.1.2 Neural network

The neural network was designed to be a scaled down version of the original one used for AlphaGo Zero. The neural network takes a game state s as its input and outputs four move probabilities p and the predicted end result v. It is composed of one convolutional layer, ten residual layers, a policy head and a value head.
Each input is a 20x4x4 binary matrix where all 20 levels represent a corresponding power of two: $power = level + 1$. For example a boolean *true* value on level 0 indicates the presence of the value 2 at its coordinates in the matrix.
The policy head outputs four probabilites p for taking each move in state s. The value head

outputs a scalar between zero and one which represents the value of the input state. A value one indicates that choosing the given move from the evaluated state will lead to a better score than the current best score and a value zero indicates the opposite.
The resulting model compiled in Keras with Tensorflow back-end has 95950 trainable and 1350 non-trainable parameters in total.

Network structure:

1. Input Nx20x4x4
2. One convolutional layer:
    - A convolution of 32 filters of size 2x2 with stride 1 and padding ″same″
    - Batch Normalization [10]
    - Rectifier non-linearity
3. Ten residual layers
    - A convolution of 32 filters of size 2x2 with stride 1 and padding ″same″
    - Batch Normalization [10]
    - Rectifier non-linearity
    - A convolution of 32 filters of size 2x2 with stride 1 and padding ″same″
    - Batch Normalization [10]
    - Skip connection from layers input
    - Rectifier non-linearity
4. Policy head - outputs probabilities vector 4x1:
    - A convolution of 2 filters of size 1x1 with stride 1 and padding ″same″
    - Batch Normalization [10]
    - Rectifier non-linearity
    - Fully connected layer with 4 hidden nodes
    - Softmax
5. Value head - outputs scalar between 0 and 1:
    - 1 convolutional filters (1x1) with stride 1
    - Batch Normalization [10]
    - Rectifier non-linearity
    - Fully connected layer with 64 hidden nodes
    - Rectifier non-linearity
    - Fully connected layer to scalar
    - Sigmoid

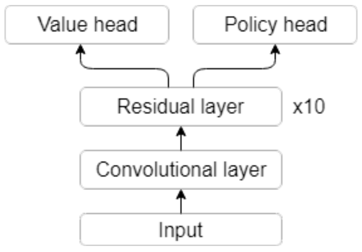

Figure 3: Structure of the neural network

3.2 Neural network with simple simulation

It is known [2] that 2048 can be played quite well with a relatively simple strategy. Simply simulate some number of random plays for each four possible moves and pick the move that had best average simulated result. With enough simulations, this approach is almost guaranteed to get 2048 tile. And it does not use any external knowledge apart from game rules.

For our second solution we simulated 7000 games using the above approach. 2000 simulations were done for each move selection (500 per possible move) and move with the best average score was chosen. Each board state was saved along with the move prediction from the simulation. Total around 15 million states were saved.
These states and predictions were used to train neural network that took state as input and produced move prediction as output.

Neural network architecture was a little bit simpler than in previous section since we only needed move predictions (only policy head was needed), also 3x3 filters were used for convolutional layers. Other than that, the architecture was similar to what we used in previous section - residual block network. The best result was achieved with 20 residual blocks with each convolutional layer having 128 filters.

Training was significantly faster than previously because we did not need to use network for simulation. The final trained network achieved performance comparable to the simulation approach.

4 Implementation and results

We implemented our solutions using Python programming language and Keras neural network framework. This choice was made since everybody was familiar with these tools and it allowed relatively fast development.

4.1 Simulator

First we needed 2048 simulator. We decided to write one ourselves since it allowed to understand the game better and also choose how to represent the board. Once we had finished the simulator it became obvious pretty fast the the speed of Python was too slow - we needed to simulate millions of games. Fortunately the solution was relatively easy, with minimal modification to the code we could use Numba [6] - a JIT compiler for Python. Numba allowed us to simulate 10 000 games with average 115 moves per game under one second (using one core).

4.2 Neural network with Monte-Carlo search tree

4.2.1 Architecture

The application is composed of three main parts: game playing simulators, the game state handler, and the state-evaluating model. The game playing simulators and the state handler each work on a separate process, which allows to utilize multiple CPU cores.
In order to take a single move, each simulator has to perform the Monte Carlo tree search using the values provided by the neural network. This means that all concurrent games continuously query the model for predictions based on the game states. Due to these requirements, evaluating one state at a time proved to be unreasonably slow. Instead of doing that, the state handler gathers game states from half of the running games and does a query to the model, which then provides the evaluations p and v for all states at once.

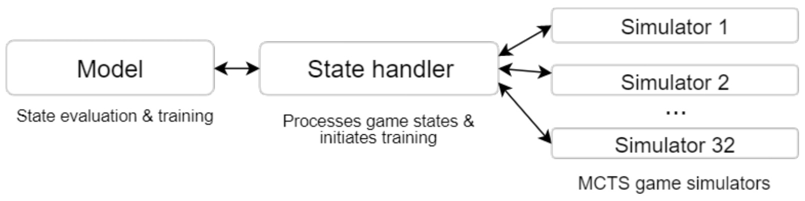

Figure 4: Training application architecture

4.2.2 Training process

The training process starts by launching the game simulator, the state handler and by initializing the model with random weights.
Each training cycle, the model is trained by using 100 batches of 2048 sample states from the 2000 previously played games. After that 25 games are played and the game states are added to the training set. If the new neural network didn't achieve a new highest mean score calculated from those 25 games, weights are restored to the pre-train state and a new training cycle is started.
The model was trained using the following loss function:

$$(p, v) = f_\theta(s) \; and \; l = (z - v)^2 - \boldsymbol{\pi}^T log(p)$$

Here v is the predicted and z is the actual value of a state and π is the actual probabilities whereas p is the predicted probabilities. Adam optimizer with the default parameters provided by Keras was used for training.

Note: All numeric parameters presented in this paragraph are the latest parameters that were to train the neural network. Various other parameter values were also tried out.

4.3 Results

The scaled down versions of the game simulation and training process were first tested on a Windows 10 PC to make sure the training process worked without bugs. The full-scale simulation and training process was carried out on the Rocket Cluster - a computing cluster with GPU capabilities in the High Performance Computing Center of the University of Tartu. The simulation used 16 cores from an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz, 20GB of RAM and one NVIDIA Tesla P100 GPU.

By starting from a randomly initialized neural network, the simulators achieved scores ranging from 500 to 4000. These scores are similar to the results from a random play.
The training was carried out for 10 hours and when using 100 simulations in the Monte Carlo Tree, 66 training cycles were performed in total. All 16 simulators combined were able to perform about 7 moves per second. Of all 66 cycles the new trained model was able to beat the neural network with the old weights four times. The best achieved score was 2638.
When using 50 simulations in the Monte Carlo Tree, 32 game simulators were able to perform approximately 40 moves per second. During 10 hours of gameplay and training, 334 training cycles were done and a new version of the initial model achieved a better median score only 2 times.
During the course of development and testing the parameters of the neural network mostly stayed the same, while only trying out a different learning rate and the amount of residual layers. Other parameters like the amount of samples the network being is trained on each training cycle, the amount of games being played to evaluate the model were also tried out to achieve better results. Unfortunately, none of the tried combinations were able to improve the model from self-play during 10 hours of gameplay and the results during the training process stayed similar to the results achieved by the randomly initialized network.

4.4 Neural network with simple simulation

More than 7000 games were simulated which produced 15 million training samples. Average final score of these simulations was 45 000.

Couple of different depth networks were tested, each training was performed with Keras default settings using 3 epochs. All used architectures were residual networks. Apart from depth, input format was varied - one variant used 4x4 input with powers of two as data, second used 16x4x4 as one-hot vectors as input. After training each network was used to simulate 100 games and the average of these results was saved. Results are presented in the following table

Residual blocks	Number of filters	Input	Average score
10	256	4x4	26 000
20	128	4x4	29 000
2	128	16x4x4	25 000
20	128	16x4x4	31 000

Table 1: Neural network comparison

As expected, increasing the network depth improved the result. Although the first and second network had approximately the same number of variables, the deeper network gave better results. Changing the input to one-hot also seemed to have positive impact as second and fourth network were otherwise identical but fourth performed better.

Performance of this simple approach was quite good with best networks achieving 70% of the initial result. Results were somewhat more stable than with pure simulation approach without the high peaks that simulation sometimes gets. Since 2048 tile usually appears when score is slightly above 20 000, the better networks managed to win the game over 90% of the time.

Training was performed on desktop workstation using NVIDIA GTX 1080 GPU.

4.5 Browser application

A browser application featuring a graphical interface was created which allows for two types of play. When auto-run is not engaged, the user can play 2048 normally and ask the model what it thinks is the best move for the given board by pressing the "Get Hint" button. [3] When auto-run mode is enabled, the model's predicted best move is always carried out automatically.

The browser application uses the user interface and framework of an existing 2048 solver. [4] In it, a pre-trained model is ran using KerasJS - for each board state, the application uses the model to predict what the next move should be, always taking the move with the highest score. [5] This does lead to very rare situations where the model wants to perform a move that does nothing. We chose to keep this behaviour to highlight the type of boards that our model does not excel at.

5 Contributions and discussion

During the project our team

- Discussed the article [12] and various other materials explaining the algorithm behind AlphaGo Zero.
- Implemented efficient 2048 simulator in Python using Numba JIT compilator.
- Implemented multiprocess reinforcement learning pipeline to train neural network to play 2048.
- Deployed the reinforcement learning pipeline on High Performance Computing Center of the University of Tartu.
- Tested the pipeline with various network architecture and MCTS parameters.
- Simulated data from 7000 self-play games using simple Monte Carlo simulation for move prediction. Used that data to train residual network to play 2048 with over 90% success rate achieving 2048 tile.
- Updated the forked 2048 browser application to use our neural network for self play [3].

We mostly achieved the project goals, understood and implemented the AlphaGo Zero algorithm. Although the reinforcement learning approach of AlphaGo Zero did not give good results for 2048, we still managed to come up with approach that uses no outside knowledge or input to train a network to play 2048 with reasonable skill.

5.1 Analysis of problems

The main reason why the network couldn't improve from self play is most likely due to the problems also noted in paragraph 2.2.2.
First of all the randomness of the environment meant that a large amount of MCTS simulations had to be used when selecting a move. In our case, to simulate games at a reasonable rate, the amount of simulations needed to be decreased from the initial 1600 to 100 or less. Decreasing the amount of simulations done for each move meant that MCTS would be more influenced by the randomness of the game during simulation, also making it harder to get good results.
Even when using 100 simulations for each move in MCTS, evaluating states on the neural network became the main performance bottleneck when simulating games. The slow iterative training process of the network also meant that evaluating different configurations took more time.
When testing out different parameters it didn't become clear which values would work best as the output of the value head. Using a scaled score meant that values in the beginning of the training process would be all very small not affecting the weights during the backpropagation. In the end the probability of beating the best last score was used, similarly to the original AlphaGo Zero paper. Outputting the scaled score was also tried but it didn't seem to alter the results.

References

[1] 2048 - Wikipedia. https://en.wikipedia.org/wiki/2048_(video_game). [Online; accessed 4-Dec-2017].

[2] 2048 AI - Using no hard-coded knowledge about the game. https://github.com/ronzil/2048-ai-cpp. [Online; accessed 10-Jan-2018].

[3] 2048 simulator with neural network. https://raulmrebane.github.io/2048zero/. [Online; accessed 10-Jan-2018].

[4] A simple AI for 2048. https://github.com/ovolve/2048-AI. [Online; accessed 10-Jan-2018].

[5] Keras.js. https://github.com/transcranial/keras-js. [Online; accessed 10-Jan-2018].

[6] Numba. https://numba.pydata.org/. [Online; accessed 10-Jan-2018].

[7] Gabriele Cirulli. 2048. https://gabrielecirulli.github.io/2048/. [Online; accessed 4-Dec-2017].

[8] David Foster. AlphaGo Zero Explained In One Diagram. https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0. [Online; accessed 4-Dec-2017].

[9] Demis Hassabis and David Silver. AlphaGo Zero: Learning from scratch. https://deepmind.com/blog/alphago-zero-learning-scratch/, 2017. [Online; accessed 4-Dec-2017].

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[12] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.