Solving Pocket Cube with Autodidactic Iteration

Piotr Mikołajczyk

1 Overview

The main goal of this project was to provide clear and transparent implementation of DeepCube - a Rubik's cube solver designed by McAleer et.al (article).

The major problem of solving optimally the cube is that we have enormous state space, sparsely connected (the graph of states is 6-regular with number of states $\approx 4 * 10^{19}$) with only one state with positive reward. The most natural way of tackling with it is to harness knowledge from the group theory. However the authors propose a new method of combining specific training of deep neural network with Monte Carlo tree search, which results in a DeepCube - a successful solver.

2 Tools used

Project was implemented in Python 3.6. Core algorithms were written in raw Python with assistance of *numpy* package for optimized mathematical operations. Plotting was realized with *matplotlib*. Serializing objects was done with *pickle*.

3 Realization

3.1 Cube model

To overcome need of computating power to train network, the problem was reduced to solving pocket cube (Rubik's cube $2 \times 2 \times 2$). Although this version is substantially easier ($< 4 * 10^6$ states, never need of more than 14 moves to solve), it was still sufficient - single MCTS without backup was having serious troubles to solve some cubes in a short time.

Setting one of the corner as immobile let reduce possible moves to only 6 (left face clockwise, back face clockwise, down face clockwise and their counterclockwise twins).

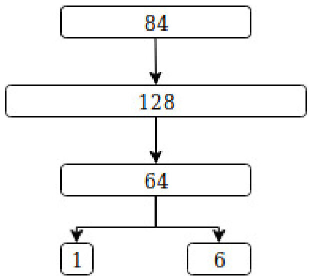
3.2 Training

The first phase of DeepCube is to train neural network to estimate both value of a state and probabilistic policy from particular cube's state. Cube's state was encoded with one-hot-encoding method to avoid any unwanted correlation with numbers.

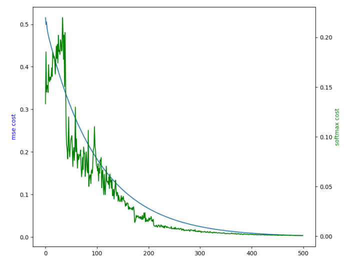
NN was divided into 3 modules: body net, policy net and value net. The output of body net was the input to the later two. The policy net was build up with softmax combined with cross entropy loss function and sigmoid activation, while the value net was given ELU activation and MSE loss function. RMSProp optimizer and vanishing learning rate were used to improve learning process. While backpropagation, the body net learns from both its children, weighting each error with respectively 0.3 and 0.7 factors.

The same observation as in the article was made regarding convergence - learning process often diverged. Weighting samples due to their distance from solved state resulted in 100% convergence.

Final network was trained in 500 rounds, each time learning on a batch of 2048 random cubes generated in a similar way that it was proposed by the authors.



(a) Neural network structure



(b) Loss function over training

3.3 Solving

When the network is trained, it is harnessed for the use of Monte Carlo tree search. Given a root state, the tree is iteratively expanded.

Each node contains information about the number of times each child was selected via tree policy, best value reached from this node after taking each child, virtual loss value for each child (for encouraging exploration) and probabilistic policy of choosing children (retrieved from neural network).

Tree policy in node s chooses move m that maximizes value:

$$\frac{cP_s(m)\sqrt{\Sigma_{m'} N_s(m')}}{1 + N_s(m)} + W_s(m) - L_s(m)$$

, where c is the exploration parameter, $N_s(m)$ is number of times when move m was chosen in state s, $P_s(m)$ is the probability of taking m in s, $W_s(m)$ is the best value ever reached after following m from s and finally $L_s(m)$ is the virtual loss dedicated to following m.

After reaching final state, the tree is reconstructed to a connected graph, in which we run breadth first search to find shortest possible path from root state to the solved one.

4 Results

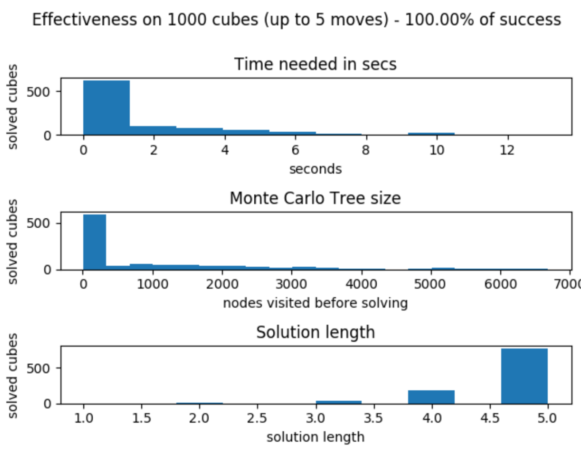
Effectiveness of the solver was measured on four groups of cubes:

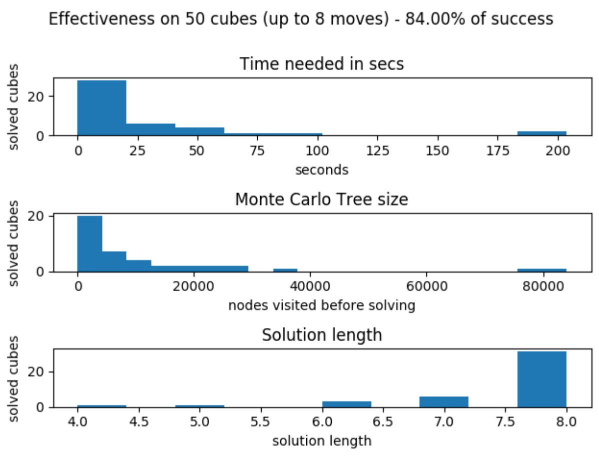
- easy cubes - up to 5 moves from solved state (easy for brute force for computer, but some of them can be still rather difficult for amateur human)
- cubes created from applying 8 random moves
- cubes created from applying 10 random moves
- hard cubes - scrambled by 30 random moves

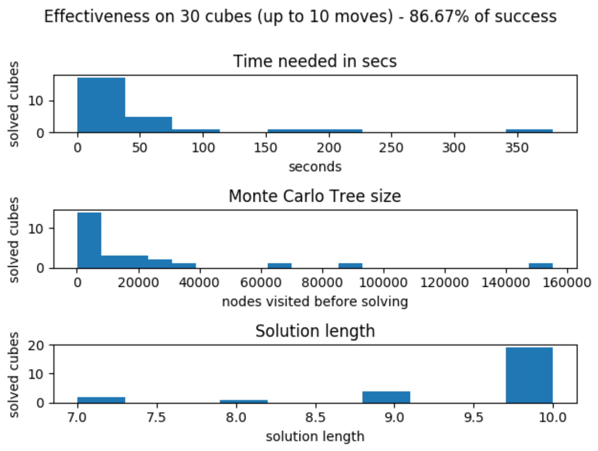
As one may expect, for easy cubes solver is fully successful. Although brute force would visit up to 7700 states ($\approx 6^5$), thanks to the hints from neural network, Monte Carlo method in general visits less than 300 states and reaches target in approximately 1 second with optimal solution length.

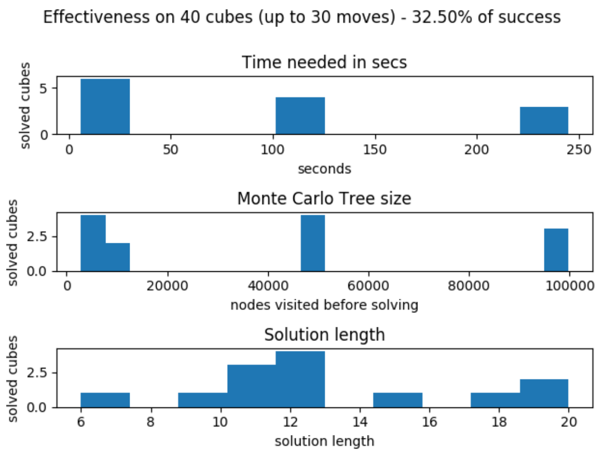
When it comes to the medium cubes, there occur some problems with timeout (solver had respectively 240 and 600 seconds for each cube). However it is still successful, solving majority of testcases. There are also some outlying cases, in which solver was completely lost and had to traverse much more of the state graph.

Hard cubes were often far too sophisticated for tree search, even equipped with neural network. Although some of them were successfully solved, there were some suboptimal sequences of moves.









5 Remarks

As one can complain about the effectiveness for the last group of most challenging cubes (which is the vast majority of all states), we should keep in mind the fact that the trained neural network was really small and the training process took less than 30 minutes exhausting learning possibility of this particular structure (see loos function plot). Undoubtedly bigger neural network and better machine for MCTS would perform erform significantly better. However it was not the point of the project to create super-efficient solver.

Regarding assumptions and goals of the project, I must say that the target was reached - there is clear, robust and easy modifiable code proving that the innovatory method called Autodidactic Iteration works well and combining Deep Learning with Monte Carlo Tree Search can result in surprisingly well self-adapting tool.