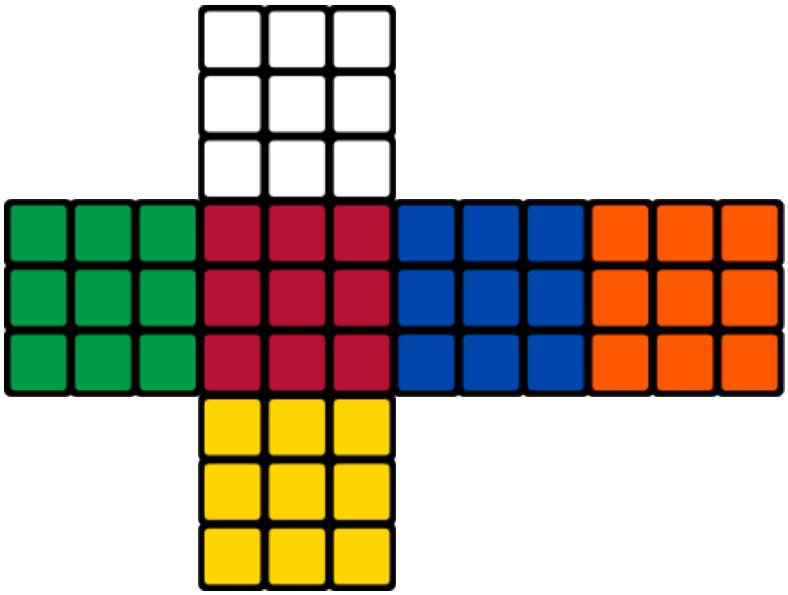
Solving Rubik's Cube with Neural Networks

Xiaotian Han, Jichun Shen, Yinzhuo Zhang



Introduction

Background

The original (3×3×3) Rubik's Cube has 8 corners and 12 edges. There are 8! (40,320) ways to arrange the corner cubes, only 7 (of 8) can be oriented independently, the orientation of the final corner depends on the preceding seven, giving 3^7 (2,187) possibilities. There are $\frac{12!}{2}$ (239,500,800) ways to arrange the edges. 11 edges can be flipped independently, with the flip of the final depending on the preceding ones, giving 2^{11} (2,048) possibilities. So, the total number of combinations is $8! \times 3^7 \times \frac{12!}{2} \times 2^{11} = 43{,}252{,}003{,}274{,}489{,}856{,}000$ [1].

Motivation

Rubik's Cube lends itself to the application of mathematical group theory, which has been helpful for deducing certain algorithms. In addition, the fact that there are well-defined subgroups within the Rubik's Cube group enables the puzzle to be learned and mastered by moving up through various self-contained "levels of difficulty". These subgroups are the principle underlying the computer cubing methods by Thistlethwaite and Kociemba, which solve the cube by further reducing it to another subgroup.

Now, there are a number of solutions being developed which can solve the cube in less than 100 moves. In 1981, David Singmaster first published his solution which solves the Cube layer by layer. In July 2010, a team of researchers working with Google, proved the so-called "God's number"(minimum number of moves to solve any) to be 20. The most move optimal online Rubik's Cube solver programs use Herbert Kociemba's Two-Phase Algorithm which can typically determine a solution of 20 moves or less.

Since all these kinds of solutions need very tough math to fully understand why they work and even more mathematic background to come up with new solutions, it’s very hard for ordinary people and even impossible for computers to come up with these solutions automatically.

With the hope of letting computer learn how to solve Rubik's Cube with some general algorithm and after carefully searching previous work online, we found that most used methods for solving Rubik’s Cube problem is reinforcement learning. But this method need so much computational power and we are not so familiar with it, so we want to find another way. Since we didn’t find any neural network related methods, we assume that it is because the objective function space is too big for neural networks to learn. But it is still worth trying since there’s no specific experiment to prove that. And our methods to solve Rubik's Cube with neural networks at least could also fill the gap and provide data for this method.

Problem Abstraction

We know that every regular Rubik's Cube has 6 faces. For every face, we can determine its move by rotate clock-wish, counter-clock-wise or rotate 180 degrees. Any possible state of a Rubik's Cube can be formed by a sequence of moves combination of its faces.

Based on this intuition, we modeled Rubik's Cube as 6 3×3 matrixes(for simplicity, we do not care about the relative location of different faces), each element of the matrix represents one of 6 colors. Then we randomly sample K element independently from all possible moves and apply these K

moves to the original Cube to get a random cube. By reversing the K moves, we get the sequence moves which can solve the random generated cube. So, our final goal is that given a starting cube state, neural network can output the next move which leads to a state that is closer to the solved state.

Note: 1. We believe that the next best move only depends on the current state, rather than recent several previous moves. So, it's more like a Markov Chain and we do not need to use recurrent neural networks to deal with these data.

2. Because of the limitation of the computational power, we are only able to implement some simple structured networks and generate quite limited number of training data points. So we set the number of max moves to 10.

Notations

- *F* (Front): the side currently facing the solver
- *B* (Back): the side opposite the front
- *U* (Up): the side above or on top of the front side
- *D* (Down): the side opposite the top, underneath the Cube
- *L* (Left): the side directly to the left of the front
- *R* (Right): the side directly to the right of the front

When a prime symbol (′) follows a letter, it denotes a face turn counter-clockwise, while a letter without a prime symbol denotes a clockwise turn. A letter followed by a 2, denotes two turns, or a 180-degree turn.

Experiment Setup

Prerequisite

- Python3
- Pip

Dependent Packages

- Keras: an open source neural network library written in Python, designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible[2].

- Numpy
- Matplotlib
- Pycuber: a Rubik's Cube package in Python3[3].
- Magic Cube: a library for the cube simulation and visualization[4].

Results and Analysis

Base Architecture

For this base model, our neural network architecture is as follow:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 16, 1, 512)        5120
_________________________________________________________________
dense_5 (Dense)              (None, 16, 1, 1024)       525312
_________________________________________________________________
dense_6 (Dense)              (None, 16, 1, 512)        524800
_________________________________________________________________
dropout_2 (Dropout)          (None, 16, 1, 512)        0
_________________________________________________________________
dense_7 (Dense)              (None, 16, 1, 512)        262656
_________________________________________________________________
flatten_2 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_8 (Dense)              (None, 18)                147474
=================================================================
Total params: 1,465,362
Trainable params: 1,465,362
Non-trainable params: 0
```

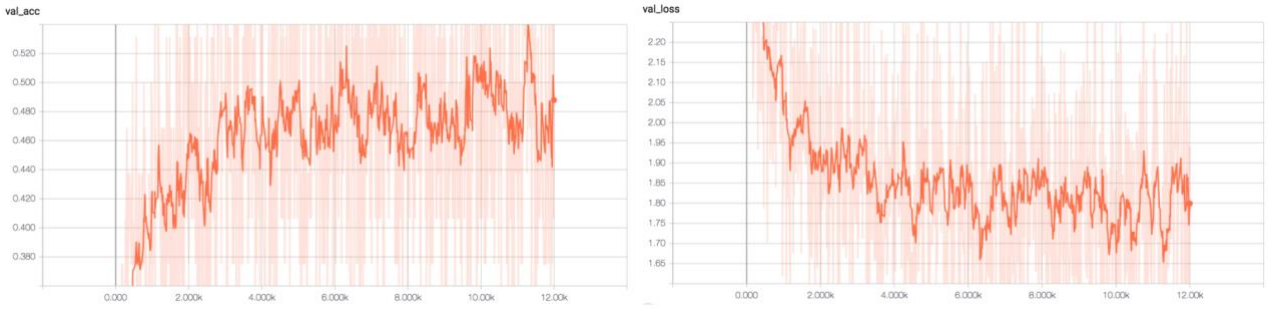
In this architecture, we first use a convoutional layer with 512 filters of size (3, 3). This layer may help us extract different patterns of a cube face. Then we add several dense layers directly, hoping to enlarge the space that our network could represents. We also use dropout among these layers to accelerate traing process and adds regularization to increase generalization. Finally, flatten precious output and use a dense layer with softmax to obtain the probability of different moves for specific output.

During training, we choose batch size as 32， number of epoch as 12000 and randomly split 20% of the input data as validation set. The activation function of all the neurals we used is ReLu and loss function is catogorical cross entropy. The optimizer we use is Adadelta. The reason that we choose Adadelta as optimizer is that we do not need to choose learning rate.

The total number of training data is 100000 with the shape of (18, 3, 1), which is just simply stack 6 faces of 3x3 matrix together. The element of the input matrix is just simple number from 0 to 5, representing 6 different colors. The labels are one-hot representation of 18 kinds of different moves.

The final result of trained model on test set(size 25000) is as follow:

```
Test score: 1.75692431927
Test accuracy: 0.490320003033
```



Data Re-representation

Last part, we use number 0 to 5 represents different colors of the cube's faces. Although it seems perform pretty well, even though we just use simple network structure, using such way to represent color indeed have some problems.

Intuitively, when we fix a viewing angle of the cube, all of the colors should be treated equally. However, if we represent colors with number, we will introduce hierarchical structure implicitly. We know that bigger number will activate neural more than smaller number. More specifically, if we use 5 represents red and 0 represents blue, our network may regard red to be more important than blue.

Based on this idea, we turn previous color representation to one-hot representation. So, our input data become (18, 3, 6) tensors, with each element as a 6 bit one-hot vector indicating which color it belongs to.

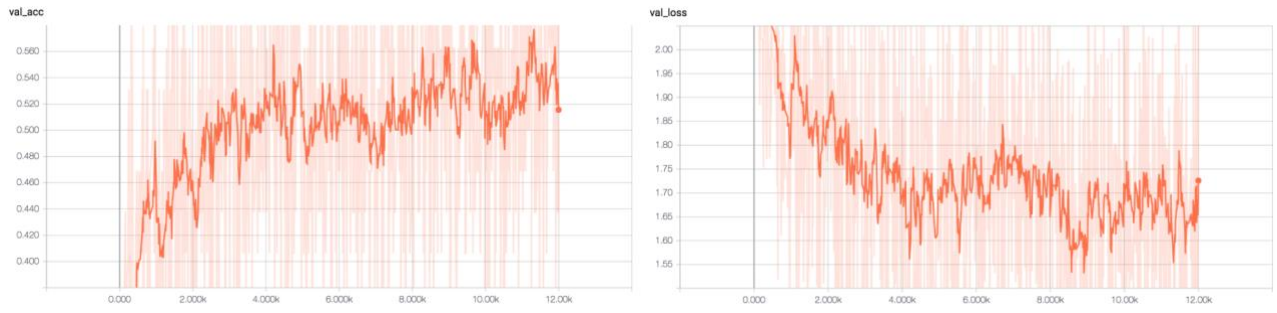
As for the architecture of our neural network this time, it is nearly the same as previous one.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 16, 1, 512)        28160
_________________________________________________________________
dense_1 (Dense)              (None, 16, 1, 1024)       525312
_________________________________________________________________
dense_2 (Dense)              (None, 16, 1, 512)        524800
_________________________________________________________________
dropout_1 (Dropout)          (None, 16, 1, 512)        0
_________________________________________________________________
dense_3 (Dense)              (None, 16, 1, 512)        262656
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_4 (Dense)              (None, 18)                147474
=================================================================
Total params: 1,488,402
Trainable params: 1,488,402
Non-trainable params: 0
```

The hyperparameters of this network is the same as previous one.

The final result is quite encouraging:

```
Test score: 1.66150145531
Test accuracy: 0.529399991035
```



We just got 3% increase in accuracy by just simply change representation of our data. This confirms that our conjecture is correct.

Changing input Data Shape

We know that 6 faces of Rubik's Cube are neighboring. However, stacking 6 3x3 face matrix into 18x3 matrix cannot preserve all of these relationships and even causes confusion when we apply convolutional layer.

So, rather than stacking 6 faces together, we decide to just keep them independent. Our input shape becomes (6, 3, 3, 6).

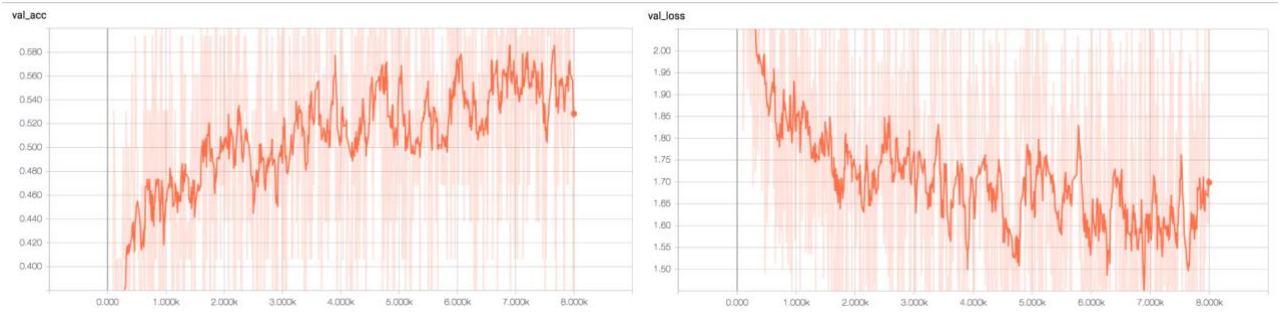
As for the architecture of our network, just simply change 2D convolutional layers to 3D convolutional layers. Then we obtain:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv3d_1 (Conv3D)            (None, 5, 2, 2, 256)      12544
_________________________________________________________________
conv3d_2 (Conv3D)            (None, 4, 1, 2, 512)      524800
_________________________________________________________________
dropout_1 (Dropout)          (None, 4, 1, 2, 512)      0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               2097664
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 18)                9234
=================================================================
Total params: 2,644,242
Trainable params: 2,644,242
Non-trainable params: 0
_________________________________________________________________
```

All the other hyperparameters are the same as before.

The final result is

```
_________________________________________________________________
Test score: 1.59777553082
Test accuracy: 0.544920003414
```



The test accuracy increases around 1.4% compared to previous results.

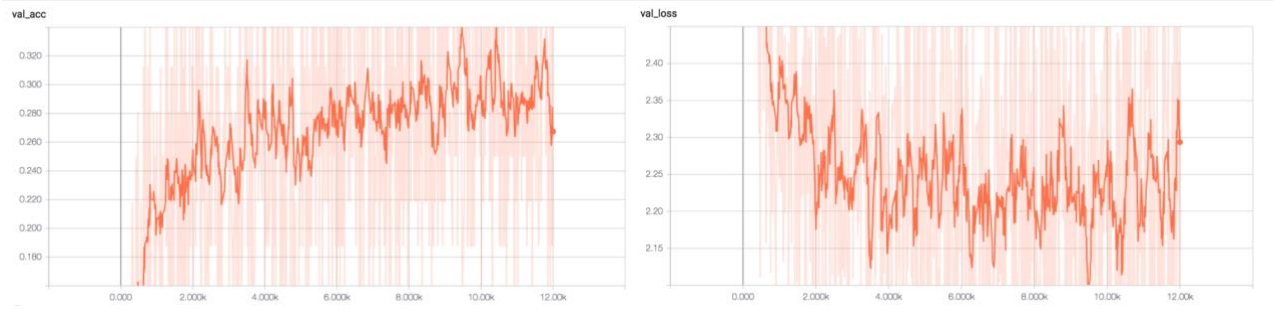
Data Augmentation

Since Rubik's Cube has naturally high color permutation symmetry, one way we can do to augment our training data is add all color permutations of the data point that we randomly generated. By doing this, our augmented data set becomes $A_6^6 = 720$ times bigger than original data set.

We use the same network structure as before. However, because of the limitation of hardware, we cannot have enough time and storage to train our network thoroughly.

```
Test score: 2.19352750778
Test accuracy: 0.295719999075
```

This result is what we got after training a whole night(around 8 hours).



From above two pictures, we notice that the accuracy and loss fluctuate heavily, which means that our network is underfitted.

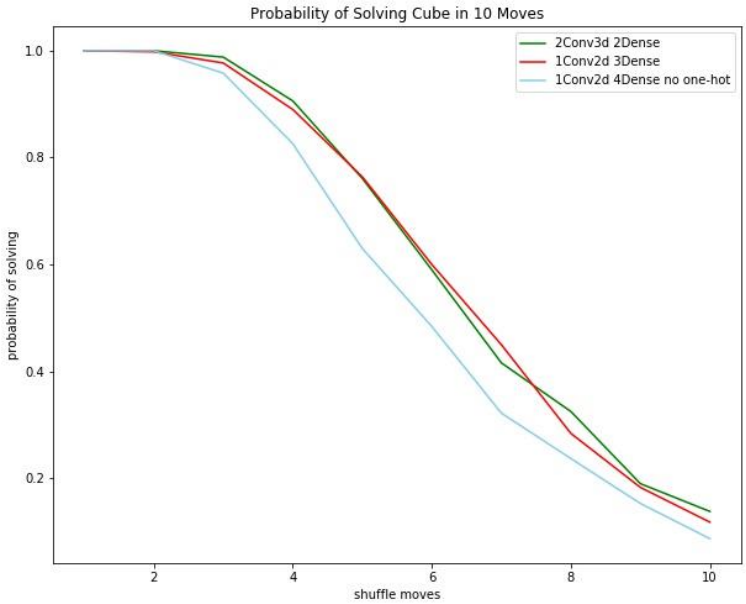
Another reason that augmented data does not perform as well as previous is that our network architecture is too simple to learn the internal symmetric relationship of the cube.

Final Restore Analysis

In the precious part, we mainly talked about several ways and improvements we made in order to increase the accuracy of next move prediction based on current state. Our final goal is to restore Rubik's Cube completely from any shuffled state.

The following picture is the probability that we restore a Rubik's Cube completely. We calculate this probability by inputting randomly generated state into our model and apply output move to the state. Repeating this process for K times and see whether it can be solved. The y-axis represents probability, the x-axis represents the number of moves that we use to shuffle the Cube. The green line comes from the model with INPUT_SHAPE = (6,3,3,6), 2 Conv3D and 2 Dense layers, convert colors into one-hot encoding. The red line comes from the model with INPUT_SHAPE =

(18,3,6), 1 Conv2D and 3 Dense layers, convert colors into one-hot encoding. The blue line comes from the model with INPUT_SHAPE = (18,3,1), 1 Conv2D and 4 Dense layers, not convert colors into one-hot encoding.



The final result is really amazing. When we shuffle 3 moves, all these 3 models could restore cube perfectly. Even though we shuffle 10 moves, all these 3 models have around 12% probabilities to restore cube. Naively, for a random shuffle, if we select random move from total 18 moves, the probability of restoring is a little bit higher than $\frac{1}{18}^{10}$, which is still quite low. So, the final result shows that our simple network indeed learned quite a lot knowledge from training.

On the other hand, red and greed lines perform better than blue line also confirm our previous assumption that there is no hierarchical relationship among colors.

Future Work

1. We notice that there are a lot of permutation invariants and equivariances in Rubik's Cube problem, which we could exploit. In the previous part, we use this to augment our training data. However, since our training data is finite and cannot cover all of the probability of Rubik's Cube state space, it is better to design a network architecture which could preserve this kind of permutation invariants. Luckily, I found a paper showing this kind of neural network[5].

2. Now our model could only solve shuffled Cube by small number of steps. In the future work, if we have better computational power, we will make more complex network so that it could work on any shuffled cube and we will also generate more training data.
3. Our model promise to give a solution for a shuffled Cube. However, in practice, people always trying to find an optimal solution with least steps. From this perspective, traditional deep neural network may be not a good choice. If we use reinforcement learning and set reward as negative value and decrease as the steps increase, this may help model find least step solutions.

Reference

[1] Wikipedia contributors. "Rubik's Cube." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 21 Apr. 2018. Web. 24 Apr. 2018.

[2] Wikipedia contributors. "Keras." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 24 Apr. 2018. Web. 26 Apr. 2018.

[3] https://github.com/adrianliaw/PyCuber

[4] https://github.com/davidwhogg/MagicCube

[5] N. Guttenberg, N. Virgo and O. Witkowski, “Permutation-equivariant neural networks applied to dynamics prediction”, arXiv: 1612.04530v1 [cs.CV], Dec. 2016.