MACHINE LEARNING ENGINEER NANODEGREE: CAPSTONE PROJECT

Aarshee Mishra

17 May 2018

1 Definition

1.1 Overview

This project is based on the domain of Reinforcement Learning. Reinforcement Learning involves an agent finding the optimal behaviour to solve the problem using a set of scalar rewards. Our agent takes one action and gets reward based upon the new state it finds itself in. The goal is to maximize overall rewards. It is very different from Supervised and Unsupervised Learning as there is no output value to minimize the loss or correlation of data that we try to find. We cannot use Supervised Learning as state space can be huge and it will be next to impossible to assign the associated action for every state that we find our self in as output value. One popular use case where we use Reinforcement Learning is self-driving cars or playing chess.

1.2 Problem Statement

This project is about trying to solve a Rubik's cube. It is arguably one of the most popular puzzle game of all time. Its most popular version is the 3×3 cube, with each box colored in one of the six different colors. Number of possible Rubik's Cube orientations are more than 4.3×10^{19}. To solve the cube, there should be only one color in each of the six different faces of the cube. Actions allowed is rotating each of the six faces as well as middle layers in one of the clockwise or counter-clockwise direction.

1.3 Metrics

The metric we will use is the fraction of the cube we are able to solve at different complexities. Here complexity refers to number of scrambles cube has gone through from the initial solved state.

2 Analysis

2.1 Data Exploration

There are nine blocks per face of the cube, so total blocks are $9 \times 6 = 54$. Now each block can have one of the six colors, so we can represent each block as a one hot vector of same size. If we now flatten the array we will obtain an array of size $54 \times 6 = 324$. This is the representation of a state in the cube. I have created an environment to simulate the cube and its states.

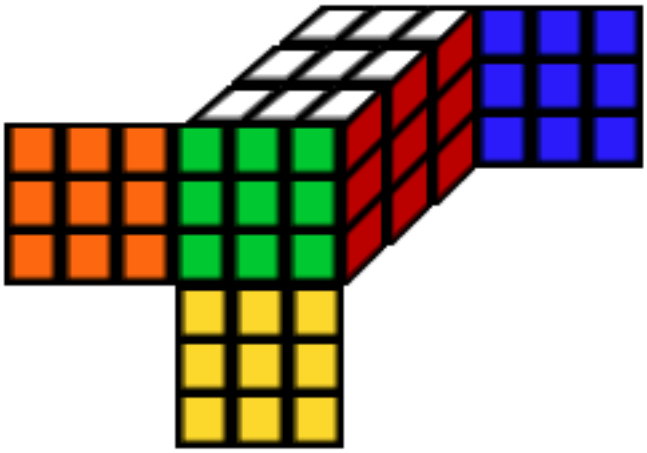
There are 4 main methods inside the environment. *Move* method is for rotating one of the faces of the cube in either clockwise or anticlockwise direction. Same method is used for solving the cube as well as in the *scramble*, which generates the scrambled cube. The orientation of the cube is also changed while

scrambling it in using *rotate*. This changes the way cube is placed without actually rotating any of its faces. For solved cube we give +100 reward, while for any other state we give a small negative reward of –1 in the *reward* method. This compels the algorithm to try solving the cube in minimum number of moves. One thing to take note of is that we are not rotating the middle layer of the cube so total possible number of actions are 12.

2.2 Exploratory Visualisation

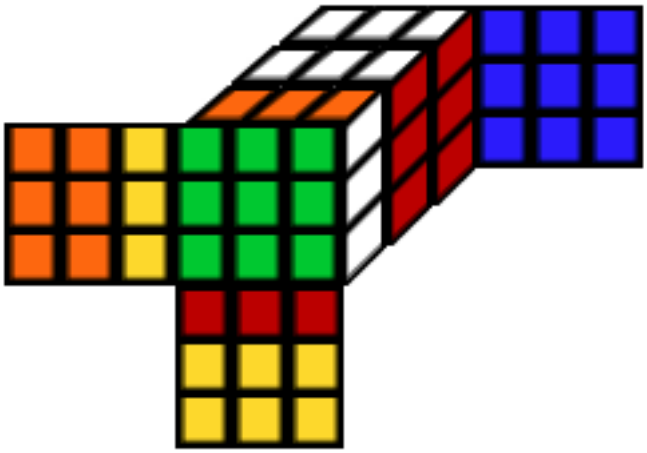
Following is the visualization of cube as well as state space representation before one hot encoding in different orientations (solved and one move from the solved cube):

COLOR CODING ≡ 0: GREEN, 1: BLUE, 2: RED, 3: ORANGE, 4: WHITE, 5: YELLOW



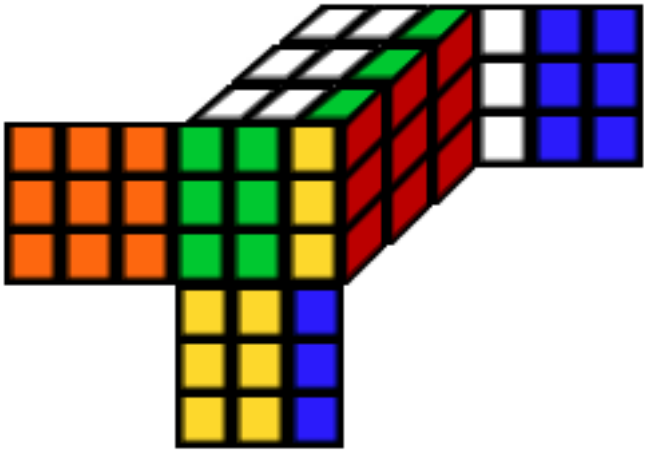
```
[[0 0 0] [1 1 1] [2 2 2] [3 3 3] [4 4 4] [5 5 5]
 [0 0 0] [1 1 1] [2 2 2] [3 3 3] [4 4 4] [5 5 5]
 [0 0 0], [1 1 1], [2 2 2], [3 3 3], [4 4 4], [5 5 5], dtype = int32]
```

Figure 1: Solved Cube.



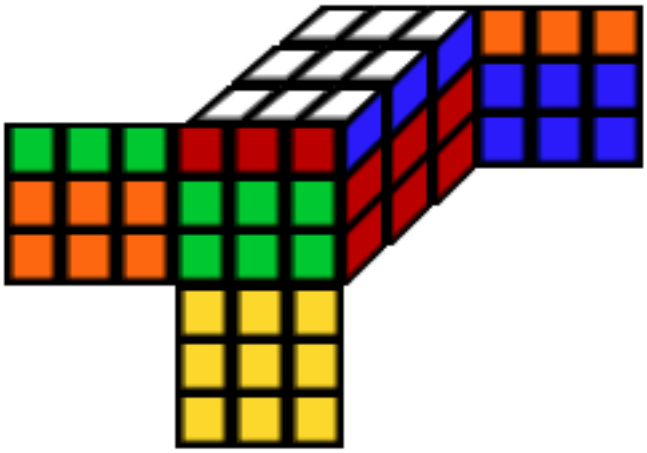
```
[[0 0 0] [1 1 1] [4 2 2] [3 3 5] [4 4 4] [2 2 2]
 [0 0 0] [1 1 1] [4 2 2] [3 3 5] [4 4 4] [5 5 5]
 [0 0 0], [1 1 1], [4 2 2], [3 3 5], [3 3 3], [5 5 5], dtype = int32]
```

Figure 2: Front Clockwise Rotation.



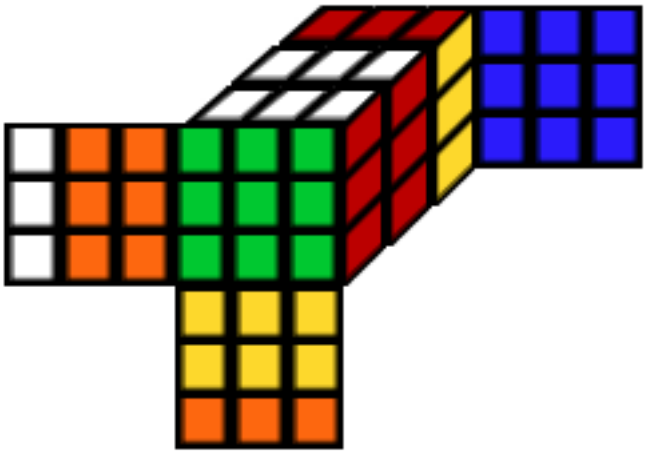
```
[[0 0 5] [4 1 1] [2 2 2] [3 3 3] [4 4 0] [5 5 1]
 [0 0 5] [4 1 1] [2 2 2] [3 3 3] [4 4 0] [5 5 1]
 [0 0 5], [4 1 1], [2 2 2], [3 3 3], [4 4 0], [5 5 1], dtype = int32]
```

Figure 3: Right Clockwise Rotation.



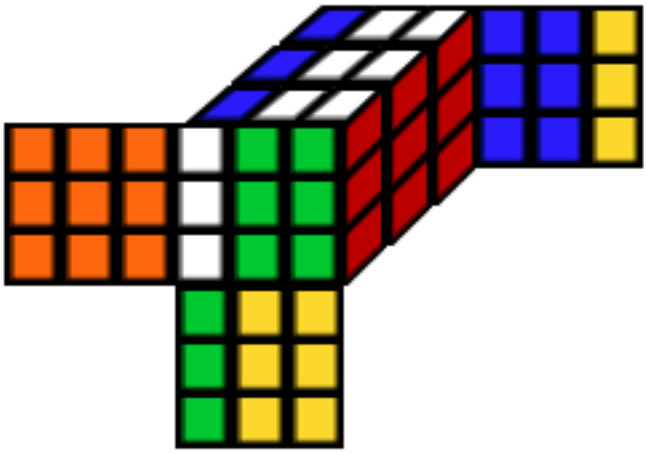
```
[[2 2 2] [3 3 3] [1 1 1] [0 0 0] [4 4 4] [5 5 5]
 [0 0 0] [1 1 1] [2 2 2] [3 3 3] [4 4 4] [5 5 5]
 [0 0 0], [1 1 1], [2 2 2], [3 3 3], [4 4 4], [5 5 5], dtype = int32]
```

Figure 4: Top Clockwise Rotation.



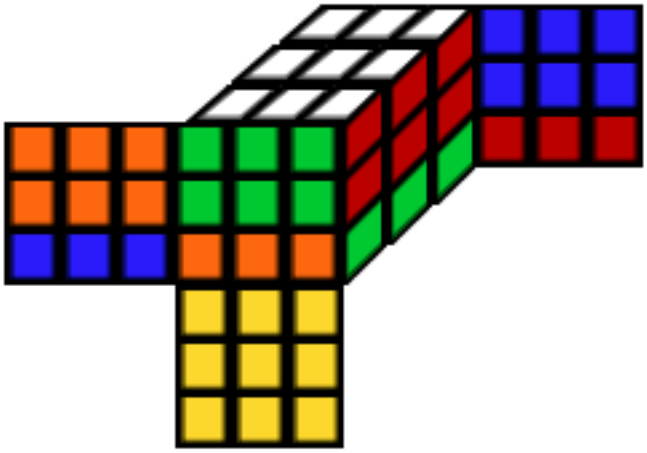
```
[[0 0 0] [1 1 1] [2 2 5] [4 3 3] [2 2 2] [5 5 5]
 [0 0 0] [1 1 1] [2 2 5] [4 3 3] [4 4 4] [5 5 5]
 [0 0 0], [1 1 1], [2 2 5], [4 3 3], [4 4 4], [3 3 3], dtype = int32]
```

Figure 5: Back Clockwise Rotation.



```
[[4 0 0] [1 1 5] [2 2 2] [3 3 3] [1 4 4] [0 5 5]
 [4 0 0] [1 1 5] [2 2 2] [3 3 3] [1 4 4] [0 5 5]
 [4 0 0], [1 1 5], [2 2 2], [3 3 3], [1 4 4], [0 5 5], dtype = int32]
```

Figure 6: Left Clockwise Rotation.



```
[[0 0 0] [1 1 1] [2 2 2] [3 3 3] [4 4 4] [5 5 5]
 [0 0 0] [1 1 1] [2 2 2] [3 3 3] [4 4 4] [5 5 5]
 [3 3 3], [2 2 2], [0 0 0], [1 1 1], [4 4 4], [5 5 5], dtype = int32]
```

Figure 7: Bottom Clockwise Rotation.

In July 2010, it was proven that maximum possible distance of a state from the solved position can be 20[4]. This number is popularly known as God's number. Below is the table listing the number of states at different distance in the cube.

Distance	Number of States
0	1
1	18
2	243
3	3,240
4	43,239
5	574,908
6	7,618,438
7	100,803,036
8	1,332,343,288
9	17,596,479,795
10	232,248,063,316
11	3,063,288,809,012
12	40,374,425,656,248
13	531,653,418,284,628
14	6,989,320,578,825,358
15	91,365,146,187,124,313
16	about 1,100,000,000,000,000,000
17	about 12,000,000,000,000,000,000
18	about 29,000,000,000,000,000,000
19	about 1,500,000,000,000,000,000
20	about 490,000,000

Table 1: Number of states at respective distances from solved state.

2.3 Algorithms and Techniques

Deep Q-Learning(DQN) is used in training the agent for solving the cube. Following is the equation of Q-learning:

$$Q(state, action) := Q(state, action) + \alpha(Target - Q(state, action)) \tag{1}$$

Here Q is the action value function and α is the learning rate. We iteratively update the action value function until it converges. Target is defined as:

$$Target = max_{action'} Q(next\ state, action') \tag{2}$$

In DQN, we try to estimate the action value function using a neural network instead of having a regular look up table. As we know that neural networks are very good function approximators. They help in approximating the action function when the state space is too large and it is not possible to enumerate the entire state space, as in this case.

We use two techniques for making the neural network converge. First one is experience learning, in this we keep a buffer of a fixed number of previous state, action, reward and next state tuples. While training our network we randomly sample a batch of such tuples instead of just taking the last value that we do in regular Q-learning. Doing so help us reduce the noise in the data and makes sure that agent doesn't forget what it has learned before. If we closely observe the target equation, we will see that neural network uses prediction from the Q-network as its target value. This leads to the network behaving erratically as gradient calculated from that value is not true gradient. To make the model more stable such that it does not change every iteration we use a different target network and update its parameter only after few iterations.

For the explore exploit dilemma I have used epsilon greedy, with constant decay rate and minimum value of 0.1.

Pseudo–Code:

```
#Constants
iteration_copy = 20
total_games = 10000
gamma = 0.99
maximum_buffer_size = 10000
batch_size = 32

Buffer = []
#Buffer Initialization
Fill Buffer with random values
epsilon = 1
total_iteration = 0
#Total Number of Games
total_games = 10000

for game from 1 to total_games:
    #Epsilon linear decay
    epsilon = maximum(0.1, epsilon - .0005)
    reset and scramble Cube
    iteration = 0
    state = Cube.get_state()
    done = false
    #We want to stop after certain iteration
    while not done and iteration < 30:
        #Epsilon-Delta
        if random sample between 0 and 1 < epsilon:
            action = random action
        else:
            #Action with maximum expected returns
            action = Q-model.predict_action(state)
        next_state = Cube.move(action)
        #Reward
        if Cube is solved:
            done = true
            reward = 100
        else:
            reward = -1
        #Appending into Buffer
        Buffer.insert((state, action, reward, next_state))
        #Buffer should not exceed its maximum size
        if Buffer.size > maximum_buffer_size:
            remove last Buffer element
        #Training
        batch = pick batch_size number of items from buffer
        loss = 0
        for each item in batch:
            #Action with maximum expected returns
            action_ = T-model.predict_action(item.next_state)
            #Target value
            if not done:
                target = reward + gamma*T-model.predict_value(item.next_state,action_)
            else:
                target = reward
            #Loss calculation
```

```
        loss = loss + (target - Q_model.predict_value(item.state, item.action))^2
    #Gradient Descent on Q-model weights
    Q_model.weights = Q_model.weights - alpha * d(loss)/d(Q_model.weights)
    iteration = iteration + 1
    total_iteration = total_iteration + 1
    #Copying Action Value into Target Value
    if total_iteration % iteration_copy == 0:
        T_model.weights = Q_model.weights
```

Apart from the constants that are defined here there are some other hyper-parameters that we are using. After some initial test cases I realized that the agent performs better if we train it step by step, similar to curriculum learning. Initially it was trained with cube scrambled for 1 to 2 times. After that again it is trained with cube scrambled 3 to 4 times. Lastly it is trained for same number of iterations with cube now scrambled 5 to 6 times. The reason of improved performance may be because it will anyway encounter states with lesser difficulty while solving for higher difficulty states. For example, if we want to solve cube scrambled X times, it is equivalent to learning how to solve cube scrambled once if we know how to solve cube scrambled X - 1 times. As we saw in the table above, the number of states increase exponentially with the increase of difficulty of the cube. This leads to declining accuracy as we move further away from solved cube.

2.4 Benchmark

There are many algorithms for solving the Rubik's cube. Earliest algorithms were human algorithms to solve the cube step by step. David Singmaster and Patrick Bossert gave their algorithms for solving the cube in 1981. Philip Marshall's algorithm can solve the cube in less than 65 moves. Herbert Kociemba's Two-Phase Algorithm can solve the cube in under 20 moves. These algorithms will be used as benchmark. My target here is to solve the cube upto 6 moves, with reasonable accuracy.

3 Methodology

3.1 Data Preprocessing

As we discussed before, we represent the cube as a vector of $3 \times 3 \times 6$ and encode it using one hot vector. After applying flattening operation on it this gives us a vector of size 324. No further preprocessing is done on the state space. We don't need to use scaling here as the data is similarly scaled as well as in binary format. Standardization of data here is not possible as the data is generated while training.

3.2 Implementation

Here is the description of all the major components of the algorithm:

3.2.1 Qmodel: *init*

This takes number of inputs, outputs, hidden layer and optimizer as parameters. This method initializes the deep Q-Network. After every layer except the last, we add *tanh* as an activation function. Optimizer is for modifying weights so that loss can be reduced.

3.2.2 Qmodel: *predict*

It takes an input tensor of size specified by the input parameter in *init* method and simply returns the predicted value.

3.2.3 Qmodel: *sample*

This takes two arguments, input as well as epsilon. It uses the predict method to predict the value on the input and used epsilon greedy strategy to return which action should be performed.

3.2.4 Qmodel: *copy model*

This method is used for copying the model weights to the target model from the current QModel.

3.2.5 Qmodel: *save model*

As the name implies this saves the current weights of the model in the given file name.

3.2.6 Qmodel: *custom loss*

This function generates the loss value which is optimized using the optimizers provided by tensorflow. Given action performed and target value as defined in (2), it calculate the error as defined in equation below:

$$Error = \sum(Target - Predicted\ Value) \tag{3}$$

Here the *Predicted Value* is:

$$Predicted\ Value = Q(state, action) \tag{4}$$

If we use gradient descent to minimize error we will end up with (1). One thing worth paying attention here is this is not true gradient as the target value used is obtained from the network itself (4).

3.2.7 Qmodel: *partial fit*

It takes input as tuples of state, action, reward and next state. It calculates the target value (2) using the second network and pass the same along with action to the *custom loss*.

3.2.8 TDmodel

This is the network we use for generating the target value. After every few iterations we copy the parameter values from the Qmodel to it. This is done so as to stabilize the gradient since we do not have true target values which we can use for training our network.

3.2.9 scramble

We pass the distance or the complexity of the cube that we want to solve as a parameter here. This gives the appropriately scrambled cube based upon the parameters we have given.

3.2.10 gen buffer

In experience replay we sample few values from the given buffer of state, action, reward and next state tuples. But at the start of the training we do not have such a buffer available. This method follows a random policy to generate the initial buffer at the start of the training.

3.2.11 play one

This is where we actually try to solve the rubik cube. It plays one episode upto maximum 30 iterations. Every iteration buffer is updated and *partial fit* is called. We return the number of iterations it took to solve the cube, total rewards and the complexity of the cube we are trying to solve from this method.

3.2.12 main

This is the main method. It uses all the methods described above for training the cube. It takes the environment, architecture of neural network, number of epochs we want to train our model for, complexity of cube we want to solve, experience replay buffer, parameters for epsilon greedy as well as initial model weights as parameter.

I plot the average of last 100 episode length as well as last 100 score. Cumulative average of last two values are also plotted.

3.3 Refinement

There are various hyper-parameters that we need to tune here.

- Architecture of Neural Network
- Optimizer
- Learning rate
- Epsilon decay
- Number of Iterations before copying model
- Buffer Size
- Batch Size
- Initial Buffer Size
- Number of maximum moves in one episode
- Rewards

In these hyper-parameters, there were parameters whose values were kept constant and as well as the ones which were varied. Reward structure was kept same i.e. 100 for solving the cube and -1 for all other states. Since the training was done only for 6 scrambles at maximum hence 30 was chosen as the maximum moves allowed in on episode. Varying initial buffer shouldn't change the results too much hence it was also kept at constant value of 100. Buffer size, batch size and epsilon decay were kept same as original Deepmind paper at 10000, 32 and constant decay upto 0.1. We tested various hyper-parameters for a total 2000 iterations with maximum complexity as 2 and compared their results. The models which performed best were trained for more iterations.

Architecture/Learning rate	0.1	0.01	0.001
$200 \times 200 \times 200$	14.9	29.5	15.2
$200 \times 200 \times 200 \times 200$	12.6	11.8	14.6
$200 \times 200 \times 200 \times 200 \times 200$	13.6	12.8	14.7

Table 2: Solved cube percentages for different architectures and learning rate using Adagrad optimizer

Architecture/Learning rate	0.1	0.01	0.001
$200 \times 200 \times 200$	12.7	14.0	12.9
$200 \times 200 \times 200 \times 200$	12.8	14.6	13.6
$200 \times 200 \times 200 \times 200 \times 200$	11.7	13.7	12.6

Table 3: Solved cube percentages for different architectures and learning rate using SGD(gradient descent) optimizer

These results are with respect to network trying to solve cube of complexity upto 2. Here $200 \times 200 \times 200$ represents the hidden layer structure. So there are five layers overall, consisting 324, 200, 200, 200 and 12 perceptrons respectively. Here 324 corresponds to input and 12 corresponds to output layer.

From these results it can be seen that Adagrad optimizer with learning rate 0.01 and $200 \times 200 \times 200$ performs the best. On further experiments it was found that this architecture does not perform well for more than 2 complexity. That is because the model is not complex enough to generalize to further complexities. By trial error it was observed that diminishing architecture with fewer neurons as we move closer to the output perform much better. Empirically the final architecture was decided as $1024 \times 256 \times 128 \times 32$.

4 Results

4.1 Model Evaluation and validation

The network was first trained for 10000 iterations on complexity 2 and again for 10000 iterations on complexity 4. Finally it was trained for 70000 iterations on complexity 6. At higher difficulty we also keep some values of lower difficulty so that network does not forget what it has learned. Results obtained are as follows:

Complexity	Success percentage
1	93.06
2	48.39
3	29.61
4	13.61
5	8.59
6	3.11

Table 4: Final results obtained

The above results are obtained on scrambled cube at various difficulty level states which our network may or may not have seen. These results are consistent and are independent of perturbations in the environment. They can also be trusted because each time we rotate a random side of a cube placed at random orientation.

4.2 Justification

As we discussed before there exist various methods for solving the rubik's cube. From the initial human based methods, such as the one developed by Patrick Bossert to the modern computer based algorithms such as Herbert Kociemba's Two-Phase Algorithm, all of them are able to solve any scrambled cube. The only difference is the number of moves they take to solve the cube. Our approach using reinforcement learning is still far away from these algorithms. Our accuracy falls sharply as we increase the number of moves of the cube.

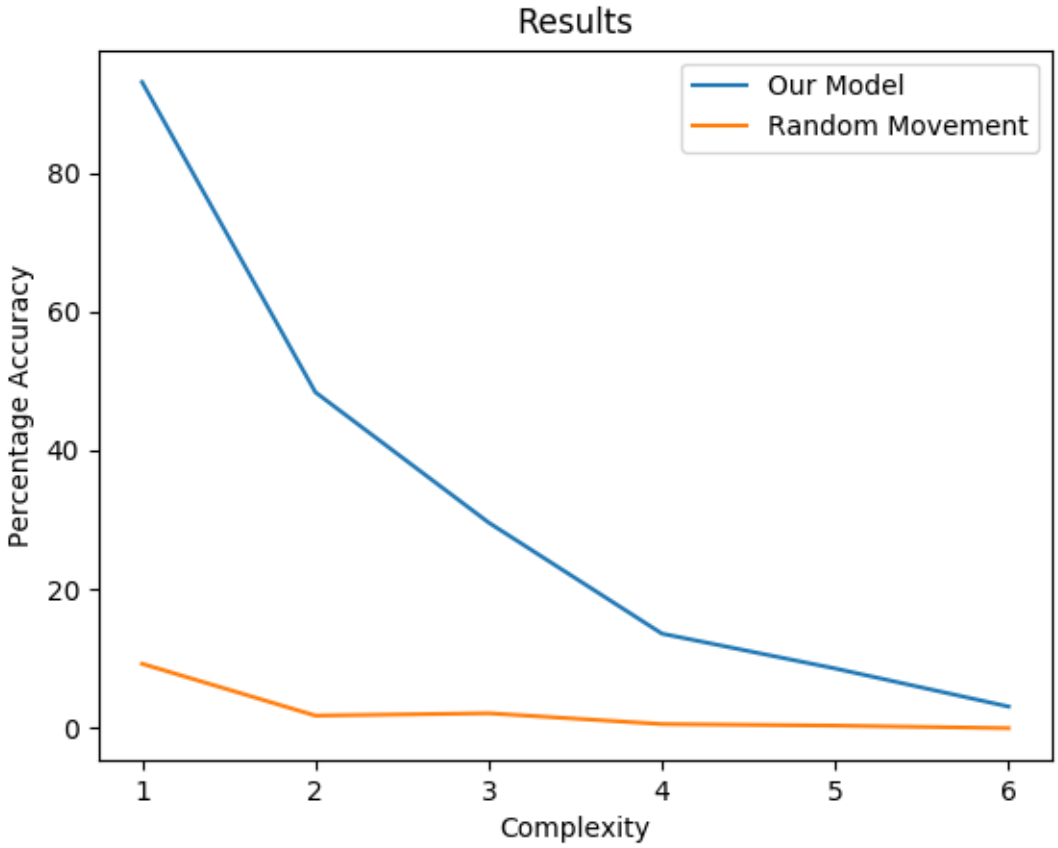


Figure 8: As we increase the complexity the accuracy falls.

5 Conclusion

5.1 Free Form Visualization

The thing which I found most intriguing about this problem is how our model learns to solve various complexities of cube. Below is the graph describing how our model learns to solve various complexities of cube as we increase the iterations

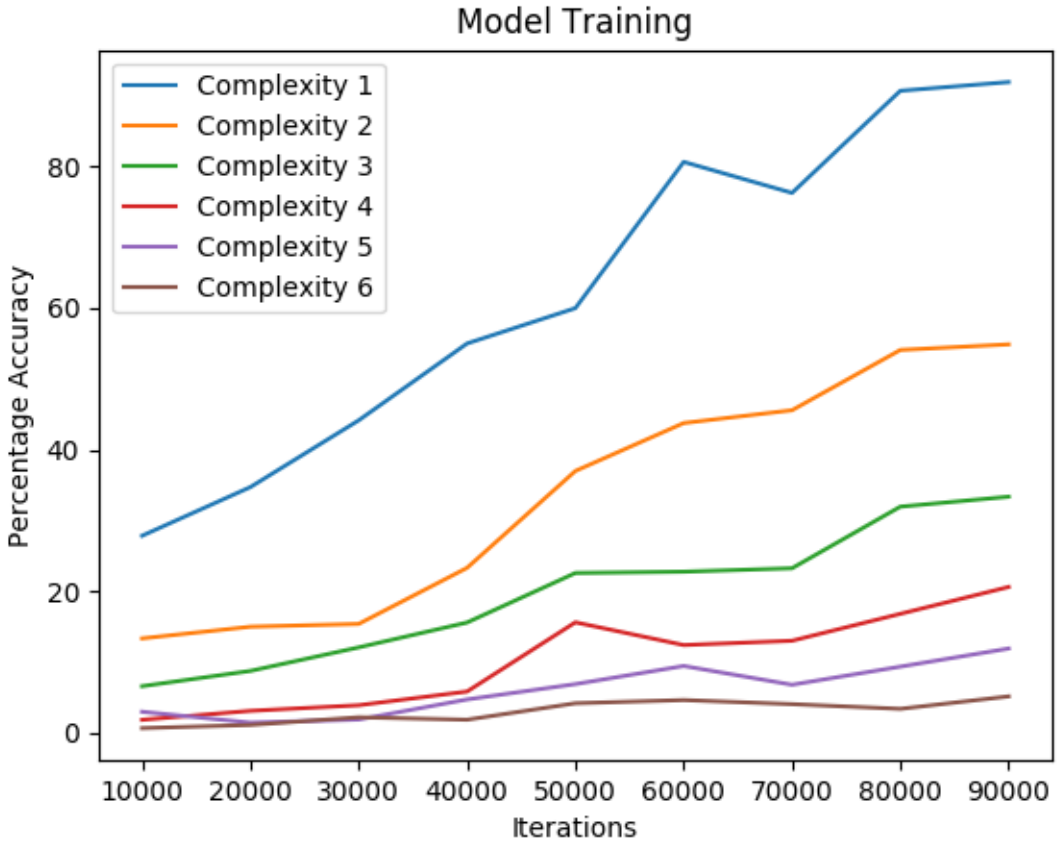


Figure 9: As we increase the complexity the accuracy falls.

From the graph it can be seen that improvement of complexities 1 and 2 is much more than others. As we saw in 1 the states increases exponentially as we increase the complexities. Our model is able to perform well for the states which have fewer states compared to others. This result is in line with our expectations.

5.2 Reflection

This project is about using deep Q-Learning to solve the rubik's cube. In regular Q-learning we maintain a table for action-value, where we store expected returns for each state, action tuple. Deep Q-learning is nothing but Q-learning with neural network for approximating value function. Directly plugging in neural network in an agent does not work. Neural networks do not converge in reinforcement learning, as the reward signal is delayed and noisy. Unlike supervised learning where underlying distribution remains same, here it fluctuates as we continue playing in the environment. In supervised learning we assume that there is no correlation between two different samples, but in this case, states are highly correlated hence dependent on the previous state.

One interesting aspect of the cube was the way to represent the cube before passing to neural network. Though I went with the naive approach of representing the cube with a vector of size 54, this could have been different as actually cube has 6 centre pieces, 8 corner pieces and 12 edge pieces which is overall 26 pieces. There are also a lot of constraints in the way they can move or be arranged.

I found writing custom loss function for Q-learning also challenging. But that does not compare with

setting the hyper-parameters of the algorithm. I found that behaviour of algorithm changes drastically with small perturbations in hyper-parameter vales. Especially coming with appropriate architecture for the learning took a lot of trial and error. I still did not reach the benchmark solution that I was trying to compete with. The main reason behind that is hyper-parameter tuning. Even after a lot of trial and error I believe that tuning is still not optimal.

References

[1] Richard S. Sutton, Andrew G. Barto *Reinforcement Learning: An Introduction.* http://incompleteideas.net/book/bookdraft2017nov5.pdf

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou Daan, Wierstra, Martin Riedmiller *Playing Atari with Deep Reinforcement Learning.* http://arxiv.org/pdf/1312.5602v1.pdf

[3] Advanced AI: Deep Reinforcement Learning in Python, https://www.udemy.com/deep-reinforcement-learning-in-python/

[4] God's Number is 20, http://www.cube20.org/

[5] Optimal solutions for Rubik's Cube, https://en.wikipedia.org/wiki/Optimal_solutions_for_Rubik's_Cube/