Machine Learning Engineer Nanodegree

Capstone Project: Rubik’s Cube Solver

Yingbo Gao

March 31st, 2017

I. Definition

Project Overview

A Rubik's Cube[1] is a well-known 3D combination puzzle. It's a cube with n*n*n small blocks, where n is how many small blocks each dimension has. Allowed moves are face rotations. The task is simple yet not apparent for the most casual player – recover the cube so that each face has the same colors. People have developed various methods to solve the puzzle. For example, reversing the permutation would definitely solve it but requires pre-knowledge of the permutation. Solving the cube layer by layer is also a good intuition to start with.

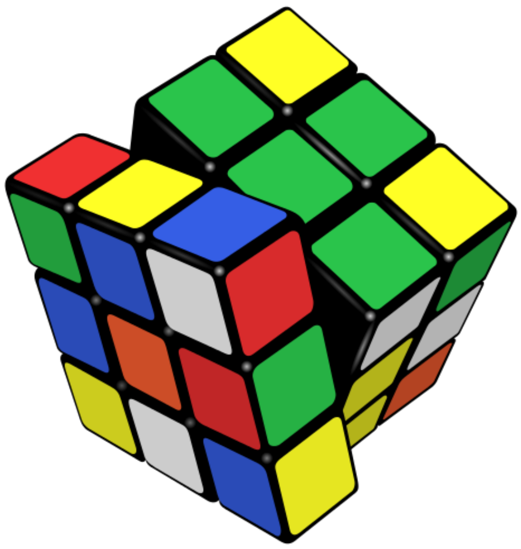


*Illustration 1: Rubik's Cube, 3*3*3*

This project looks at the puzzle from a machine learning point of view, more specifically, a reinforcement learning point of view. If a Rubik's Cube is defined for a learning agent, can it learn to solve the puzzle? Will it come up with a different way of solving it than humans? Or will it just simply remember(search) all the possibilities of permutations and become a Rubik's Cube go-to dictionary?

The project starts out by implementing a 2*2*2 cube. Choosing n=2 is a rational choice, because the so-called pocket cube[2] variant has 3,674,160 possible positions, while a standard 3*3*3 cube has 43,252,003,274,489,856,000 possible positions. A lower-order variant is good for the purpose of examining the possibilities of using a reinforcement learning agent, and not losing the generality of higher-order variants. A Q-Learning agent is implemented and tested on this 2*2*2 cube. Since the state-action space is huge even for a pocket cube, function approximation with a neural network is also tried. The data is just the cube itself, when the agent sees a position and decides on an action,

1 https://en.wikipedia.org/wiki/Rubik%27s_Cube
2 https://en.wikipedia.org/wiki/Pocket_Cube

the cube’s faces will be rotated accordingly, and give a new position for the agent to learn from. Tests will be done with randomly permuted cubes.

Problem Statement

The task of the project is: given a 2*2*2 Rubik’s Cube implementation, with which cube’s states and allowed actions are well-defined, train and test machine learning agents that can solve randomly permuted cubes.

In order to achieve this, the classic Q-Learning algorithm[3] is used, where an agent observes the states of a cube and performs actions on it. The letter Q denotes Q-Values, which in short, are scores associated with a certain action at a certain state. The job of the agent is to maintain and improve a table called Q-Table, in which state-action Q-Values are stored. The formula for Q-Learning is as follows:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Illustration 2: Q-Learning formula

where the agent updates a Q-Value towards a learned Q-Value with a learning rate alpha. The learned value can be seen as the sum of an instant reward of taking the action at the state and a discounted estimate of future values at the corresponding next state.

After training and a Q-Table is learned, in test time, the agent just chooses the actions that gives the highest Q-Value at current state, and keep on until the cube is solved or when it reaches the maximum action-count limit.

Note that Q-Table can be very expensive and inefficient to learn and store, a modified version of Q-Learning with neural network function approximation is also examined. Simply put, the Q-Table is nothing else but a mapping from a state to an action, we can try to learn a neural network that takes states as input and output Q-Values for all the allowed actions. The task is then training the weights in the neural network and in test time, take actions that the neural network outputs highest Q-Values for. See illustration below[4] for the neural network structure:

3 https://en.wikipedia.org/wiki/Q-learning
4 https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

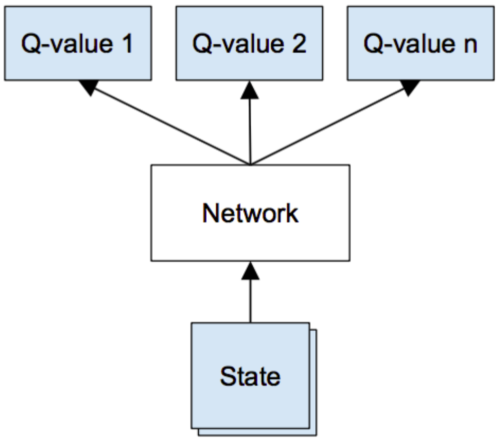


Illustration 3: modified Q-Learning with neural network function approximation

Since the state space is so large (about 35 CPU years spent to find the god-number to be 20[5]), the test for this capstone project is formulated this way:

After training, for permutation moves from 1 to 20, give the agent n (n=100 during training iterations for faster testing, n=10,000 for final testing) number of correspondingly randomly permuted cubes for it to solve, limit the maximum recovery moves to be 20, examine recovery ratio and averaged recovery moves. With my personal laptop's computing power, I don't expect an agent learning the cube perfectly, rather, I expect to see how the agent's knowledge for solving the cube evolves and improves (hopefully) overtime, and to what extent is this machine learning approach appropriate for solving similar tasks.

Metrics

As described above, the test for the Q-Learning agent will be done with randomly permuted cubes with different permutation moves, an agent is considered to be better when it has higher recovery ratios and lower average recovery moves.

Similar tests will be done for the modified Q-Learning with a neural network function approximation too. The test results of this modified learner will be compared with the classic Q-Learning agent.

Manual inspection is also possible with the help of this website, where a visual implementation of a pocket cube is provided. The goal of manual inspection is to check the validity of prior tests and also compare the agents' performances against human intuitions.

5 http://www.cube20.org/

II. Analysis

Data Exploration

Since in this project the data comes from the 2*2*2 pocket cube implementation itself, this part describes the implementation:

		16	17				
		19	18				
8	9	0	1	12	13	4	5
11	10	3	2	15	14	7	6
		20	21				
		23	22				

Drawing 1: pocket cube facelets' indices

For a pocket cube, there are 6 faces and each face has 4 facelets. Drawing 1 shows the indices for these facelets. Note that indices increase with F(ront), B(ack), L(eft), R(ight), U(p) and D(own). Also for each face, the indices are arranged in a way that if you look at the face from its above, the numbers increases clockwise. This way, a state of the cube is just a vector of length 24, in which color codes are stored. Another advantage of this is that, for allowed rotations (F,B,L,R,U,D and f,b,l,r,u,d, where letters denote faces, uppercase means clockwise if looked from above, and lowercase means counter-clockwise), the implementations are just cyclic rotations of corresponding vectors, illustrated by Drawing 2:

	19	18	
9	0	1	12
10	3	2	15
	20	21	

	17	16	
13	4	5	8
14	7	6	11
	22	23	

	16	19	
5	8	9	0
6	11	10	3
	23	20	

	18	17	
1	12	13	4
2	14	15	7
	21	22	

	5	4	
8	16	17	13
9	19	18	12
	0	1	

	3	2	
10	20	21	15
11	23	22	14
	6	7	

Drawing 2: Inner and Outer facelets corresponding to rotations on each face

For easier use, class methods that check if each face contains a single color (a "good" cube), if two cubes are equal, and allow a string as input that denotes a sequence of rotations are implemented. Face colors are coded with their initials (green, blue, orange, red, white and yellow). For the Q-Learning agent, states are defined as a string formed by joining the color codes from index 0 to 23. Actions are 90 degrees clockwise/counter-clockwise face rotations denoted by F,B,L,R,U,D,f,b,l,r,u,d. Q-Table is stored in python as state dictionary of action dictionaries. For the modified Q-Learning agent with a neural network, the difference is that state are coded as 144-bit long vectors first, this is because there are 24 facelets, each has 6 possible colors, using a one-hot coding, we get the vector. This avoids the ambiguity of using values from 0 to 5 to code the six different colors.

Exploratory Visualization

With the help of an online 2D visualization provided by ruwix[6], it is easy to manually examine the cube implementation, gather some intuitions and get an idea of how difficult the task might be.

A cube will return to its initial condition after multiples of 360 degrees of a same face turn.

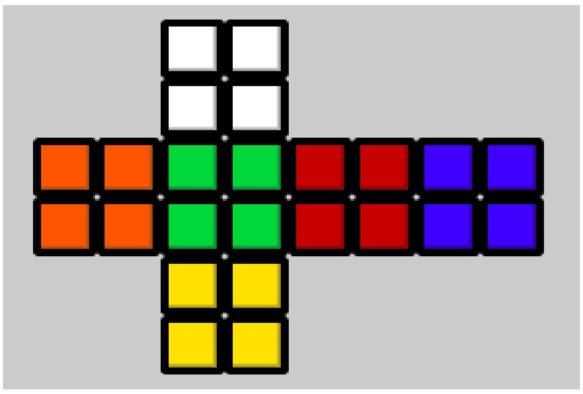


Illustration 4: Original

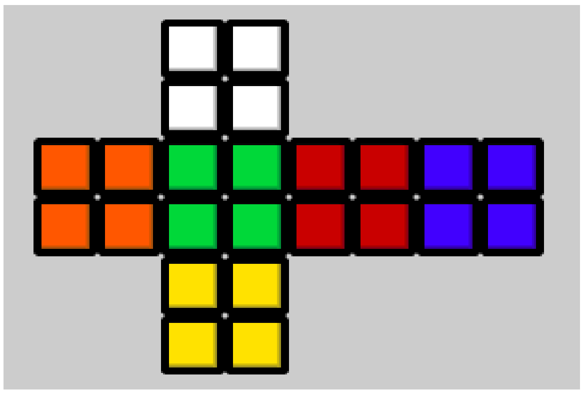


Illustration 5: After ffffRRRRRRRR

A cube will be good if rotated, but not the same as original.

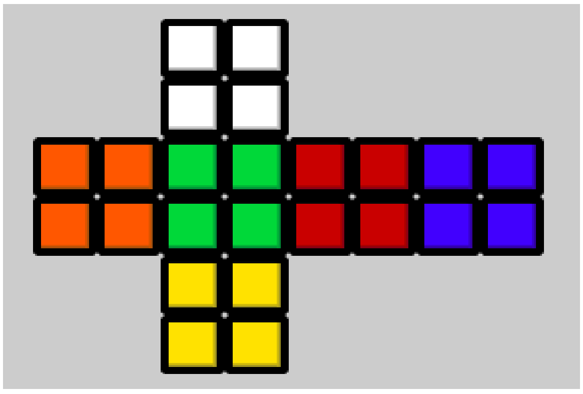


Illustration 6: Original

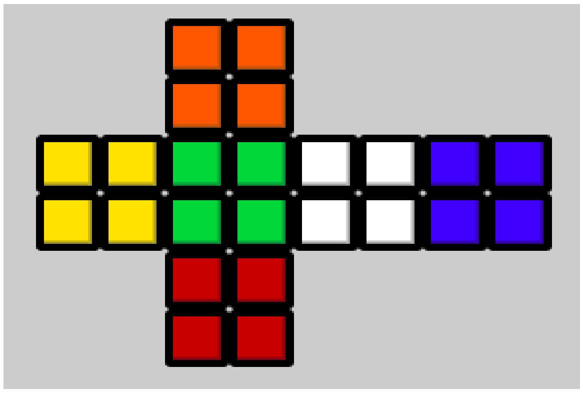


Illustration 7: After Fb

6 https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php

The order of face rotations matter.

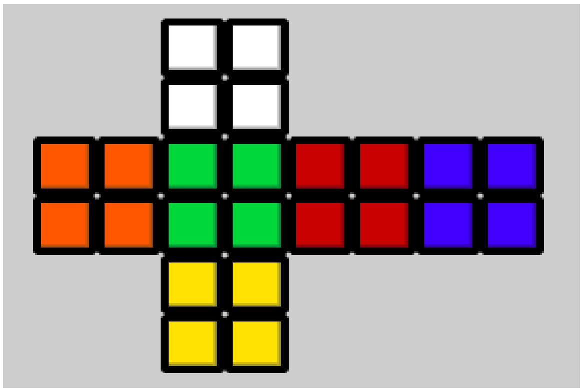


Illustration 8: After FRrf

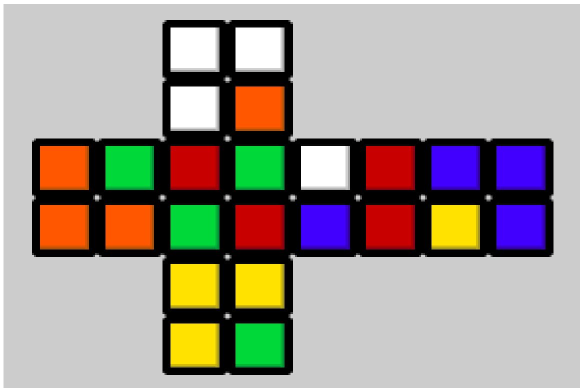


Illustration 9: After FRfr

One more observation is that after 5 random face rotations, the cube usually end up in a position that's not very intuitive for human to solve at a first glance.

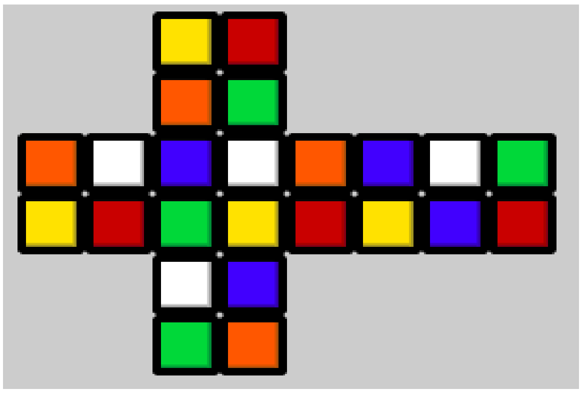


Illustration 10: After FRUBL

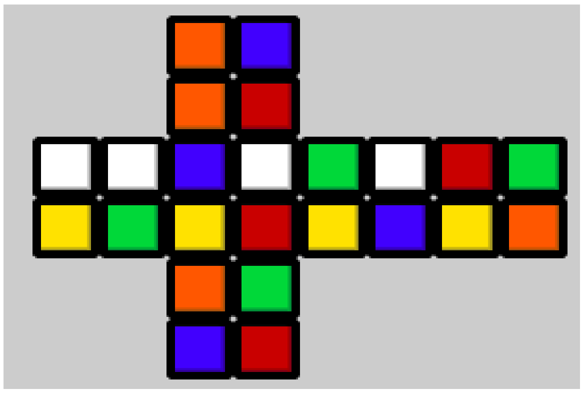


Illustration 11: After dlbur

Algorithms and Techniques

The Q-Learning Agent

The update rule is shown again in Illustration 12, there are three things that need to be defined: learning rate alpha, reward r and discount factor gamma. For reward, since we only care about if an agent can recover a cube to a good state, only when the next state is good, the reward is set to be 100, otherwise the reward function will give 0. One can argue we should reward more for positions that have less permutation distance from a good state, but I'm more interested in what the agent does if it's only goal is to recover, without any knowledge of whether one position is easier to solve than another. When initializing the Q-Table, I set values to be 0, this is because only a good position should propagate positive values out, also this gives a way to shrink the size of the table down. With this definition of reward and initialization, alpha is set to be at 1, which means the Q-Value of a state-action pair will only depend on its instant reward and its estimated future value. For gamma, I set it to 0.5, since the instant rewards can only be 100 or 0, and Q-Values are initialized to be 0, this is just a discount for future reward.

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Illustration 12: Q-Learning update rule

In training, each round will have 20 permutation lengths and 1000 training examples for each permutation length. For each example, the agent is allowed to explore/exploit only 5 moves, this is because most of the states it ends up in has no non-zero Q-Values for it to propagate to other states, there's no point for it to try more positions. The decaying function for the exploration-exploitation factor is simply a linear function that decays from 0.5 to 0.1. In testing, each round will have 20 permutation lengths and 100 testing examples for each permutation length. After 100 train-test rounds, a test with 20 permutation lengths and 10000 examples each permutation lengths will be done. Note that during testing, an agent is allowed to try maximum 20 moves.

The modified Q-Learning Agent (with a neural network for function approximation)

The update rule is the same as the the Q-Learning agent with alpha=1 and gamma=0.5. Then the algorithm can be written as[7]:

```
initialize replay memory $D$
initialize action-value function $Q$ with random weights
observe initial state $s$
repeat
    select an action $a$
        with probability $\varepsilon$ select a random action
        otherwise select $a = \text{argmax}_{a'} Q(s, a')$
    carry out action $a$
    observe reward $r$ and new state $s'$
    store experience <$s$, $a$, $r$, $s'$> in replay memory D

    sample random transitions <$ss$, $aa$, $rr$, $ss'$> from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then $tt = rr$
        otherwise $tt = rr + \gamma\max_{a'} Q(ss', aa')$
    train the Q network using $(tt - Q(ss, aa))^2$ as loss

    $s = s'$
until terminated
```

Illustration 13: Deep Q-Learning Algorithm

7 https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

Replay memory D, neural network QNN, minibatch sampling, loss function and neural network training should be explained. Replay memory is a way to avoid the neural network to converge to local optimum or “overfit” to recent experiences. The idea is just to store all past experiences and sample from them to train the neural network, which makes training more similar to a normal classification or regression task. In my implementation, it’s done by saving past state, action, reward, next state information into a Mem class, in which store and get methods are defined. The neural network has input dimension of 144 (one-hot state vectors), output dimension of 12 (actions). It has 2 hidden layers, with 96 and 48 nodes. Because the instant rewards can only be 100 or 0, I chose “relu” as activation function for each layers. The nodes are fully connected. Minibatch sampling is done by randomly get user-defined number of past experiences at each time step for the neural network to fit. Loss function is defined as mean squared error between the neural network output and the same output with the chosen action’s output replaced with desired Q-Value. For training, I used stochastic gradient decent with a learning rate of 0.001. Momentum is disabled because two adjacent training examples are randomly sampled.

During training the convergences for 1 action permutation cases are first monitored. Then the train-test round settings are identical to the Q-Learning agent, with number of rounds reduced because of too much training time. Note that for both agents, testing information in each round is stored for later visualization.

Benchmark

As shown earlier in the exploratory visualization part, 5 random face rotations will usually end up in a position not very straightforward for human to solve. Also, when permuted even more, the complexity for my eyes don’t seem to increase much. Thus, recovery rate at permutation length 5 is a good measure of how good an agent is. Note that for a Q-Learning agent, once it gets a good state-action Q-Value judgment (be it Q-Table or neural network), it’s very fast for it to solve (or fail to solve) a new given state. In other words, if both the agent and I know how to solve a position, the agent will take much less time. Thus, at least for me, if an agent can solve permutation length 5 cubes with a high recovery rate, I consider it to be good. One clarification has to be made though: for my eyes a permutation length 5 cube is not that easier than a permutation length 100 cube, and 200 doesn’t seem much harder than 100. This means when a human try to solve a permuted cube, he/she doesn’t solve it by remembering all simple permutations, instead, he/she tries to solve the cube on a higher (or more abstract) level. Which “layer” should I recover first? Maybe try recovering this face first? Why is this facelet here? These are the questions a Q-Learning agent will never ask itself. Therefore 5 is an empirical measure, a much better benchmark would be giving all positions to the agent for it to solve, and compute the recovery rate in the whole state space, but limited with my laptop’s computing power, I decide to stick with 5, the empirical number.

III. Methodology

Data Preprocessing

No data preprocessing is needed because the reinforcement learning agents are trained by explore-exploit in the state space. All date comes for the agents' experiences.

Implementation

The Cube Class

As described earlier, the cube class is implemented with facelets representations. An alternative solution would be to define cubelets with their positions and orientations. The reason behind choosing the facelet implementation is that it's easier to convert facelets' color information directly to vector states, and also it's more straightforward since what the agents look at is only the faces, not the 3D structure of the cube.

The Agent Class (Q-Learning)

Apart from the core update rule, trainData and testData lists are maintained, a qclean method is implemented to decrease memory usage, a solve method is defined, a sinlgeTest method and a test method are coded too.

The trainData and testData lists are for more comprehensible work flow. In each Train-Test rounds, the two lists are first cleared and then generated for the agent to access.

The qclean method cleans up the qtable by deleting states that have only 0 values for all actions, this is for the purpose of saving memory and also avoiding useless 0 Q-Value propagation.

The solve method just chooses the action that has the highest Q-Value at a certain state. When the state is not in the Q-Table, it's either because the agent hasn't seen the state, or the state has been visited before but no useful information (all 0 Q-Values) was learned and thus cleaned up by qclean method. In this case, the solve method just takes a random action.

The singleTest method is for manual inspections. It accepts a string of permutation actions and calls for the solve method.

The test method maintains testCounts, recoveryCounts and solveCounts, and return these lists for later use.

The Agent Class (modified Q-Learning)

A QNN class and a Mem class are coded for modified Q-Learning. A piece of code that monitors the neural network output at each time step is essential for a simple check of convergence. Some support codes that convert the string states to vectors for neural network are also important.

With the help of keras[8], a simple QNN class with setup described in the Algorithms and Techniques part can be achieved. The fit method and the predict method are relevant for the modified Q-Learning agent.

The Mem class just maintains four lists of states, actions, rewards and next states. For the store methods, a check on if the state-action pair has been seen before is done, because first duplicate storage waste memory, also more entries for a certain state-action pair in the Mem means more chance of getting sampled. The get method returns a dictionary of the sampled s, a, r, ns information.

By importing sys and using the stdout, I managed to monitor in the terminal how the neural network’s output for a certain cube position changes overtime, this is very nice because I can manually check the convergence for any state, and analyze the outputted Q-Values.

Training data is formatted to be numpy vectors for the QNN the fit or predict on.

The ProgressBar Class

Since the training and testing may take a long time, I implemented a ProgressBar class which basically writes out percentage of finished work in a slightly prettier way.

The Train-Test Rounds

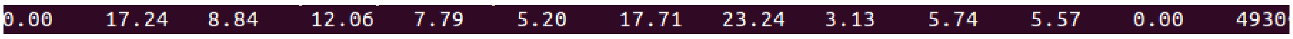
These codes just manage the Train-Test loop logic, saves test results to files and print out relevant information for monitoring.

Refinement

When I was prototyping the Q-Learning agent in my Windows system, after several hours of training, I ran into memory error. This gives the intuition of the qclean method in the Q-Learning Agent class. By cleaning up states that have all 0 values, memory usage can be saved, also the size of the Q-Table becomes relevant for knowing how much the agent knows about the state space.

Another major change is made in terms of the neural network setup. At first I set the minibatch size to be 1 (sample 1 random experience from replay memory for fitting) for faster training. But through testing and manual inspection I found that the network doesn’t always converge even for the simple case of only permuting the cube with a “b”. Later I set the minibatch size to be 10 and the neural network start to converge more constantly after some training.

Below is the neural network’s output with minibatch size 1 after 4930 time steps.

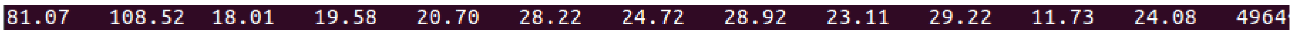


```
0.00    17.24   8.84    12.06   7.79    5.20    17.71   23.24   3.13    5.74    5.57    0.00    4930
```

Illustration 14: QNN output for "b", minibatch size 1

8 https://keras.io/

The first 12 numbers correspond to actions = ['F','B','L','R','U','D','f','b','l','r','u','d'], it’s apparent that we expect the QNN outpus Q-Values near 100 for “F” and “B”, but these two values don’t seem to converge even after longer training. Setting minibatch size to be 10 has helped to solve the problem:



```
81.07    108.52  18.01    19.58    20.70    28.22    24.72    28.92    23.11    29.22    11.73    24.08    4964
```

Illustration 15: QNN output for "b", minibatch size 10

One intuition is that bigger minibatch size means more past experiences during one fitting round. The QNN is more robustly fitting past experiences without overfitting to new samples each round.

IV. Results

Model Evaluation and Validation

The Q-Learning Agent

The following table shows the final testing result after 100 train-test rounds for the Q-Learning agent:

PL	1	2	3	4	5	6	7	8	9	10
RR	100	100	100	99	89	76	65	54	44	35
AM	1.00	1.67	2.35	3.11	5.40	7.75	9.92	11.78	13.47	14.83
PL	11	12	13	14	15	16	17	18	19	20
RR	29	23	19	15	12	10	8	6	6	4
AM	15.80	16.70	17.30	17.85	18.25	18.61	18.85	19.13	19.16	19.36

Table 1: final test result, Q-Learning, PL=permutation length, RR=recovery rate in percent, AM=average moves to recover

Below is the Q-Learning agent's recovery rates for permutation lengths = [1,2,3,4,5,7,9,12,16] over 100 Train-Test rounds:

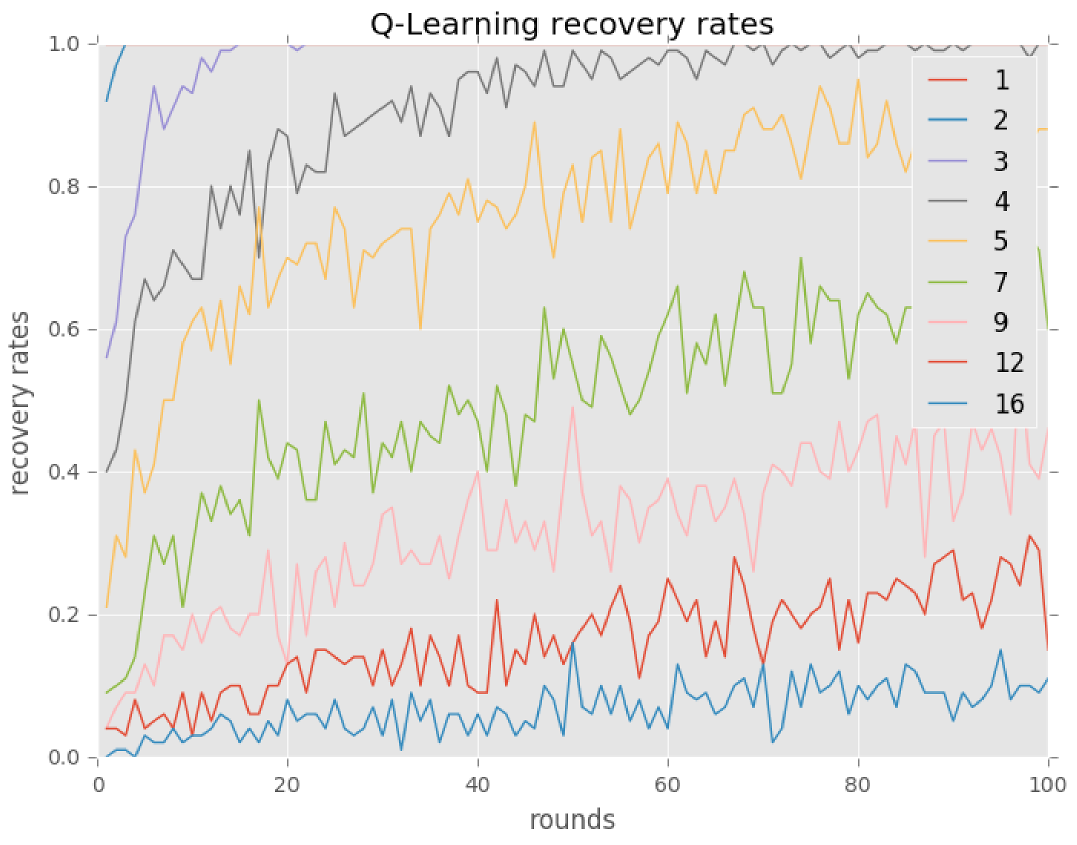
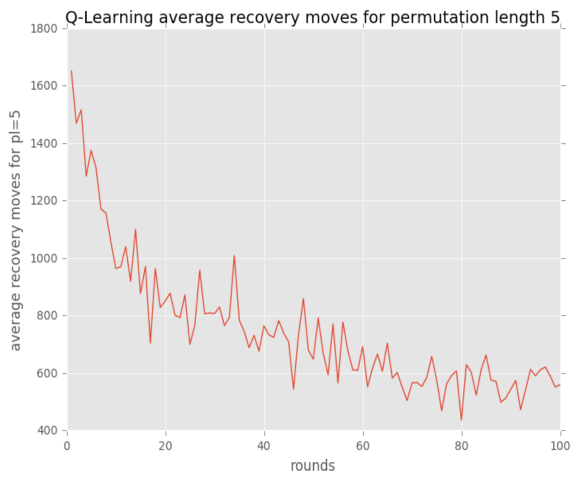
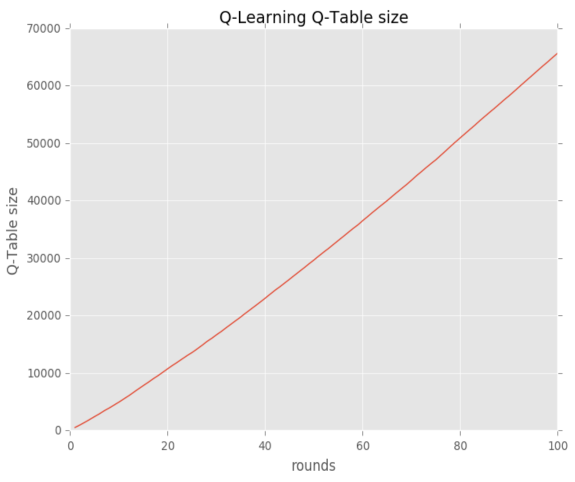


Illustration 16: Q-Learning agent recovery rates overtime

It can be seen that the recovery rates do go up, and more interestingly, the ability for the agent to solve a given permutation length (a hard position) tends to go up once it learns how to solve positions with shorter permutation lengths (an easier position). It makes sense because the Q-Learning update rule together with the 100-0 reward definition and 0 initialization means that the agent can only “learn” by propagating Q-Values from easier positions to harder positions.





The two illustrations above show how the average recovery moves go down overtime and how the Q-Table size increase more or less linearly. The number of possible permutation positions go up exponentially (neglecting duplicate positions), at least for permutation lengths up to a certain point, it’s interesting to see that even with a linearly increasing Q-Table size the agent manages to solve rather complicated positions.

Below shows some manual inspection results:

Permutation	Solve Moves	Solved
F	f	True
d	D	True
Fu	Dl	True
dl	LD	True
LBd	Dfu	True
DDD	D	True
FRUBL	lfluf	True
dlbur	DLBLD	True

Several points to note:

- It recovers F with f, d with D, dl with LD, which are direct reverse actions
- It recovers Fu with Dl, DDD with D, which means that the agent also finds alternative ways to recover the cube to a good state

- For more complex cases FRUBL and dlbur, the agent solves it with ways that are not instantly obvious for me, but after manually checking them with the help of ruwix[9], it does fully recover the positions. Note that both the recovered positions are no the original cube

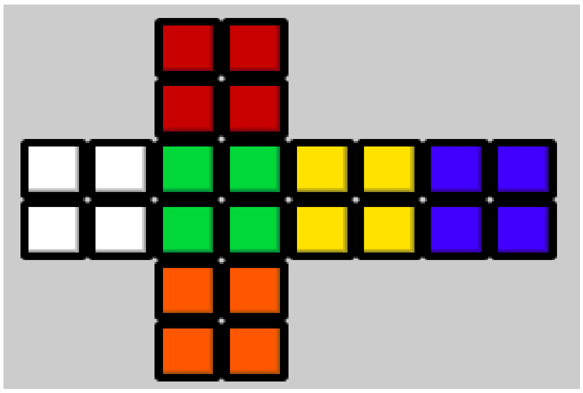


Illustration 17: 'FRUBL' + 'lfluf'

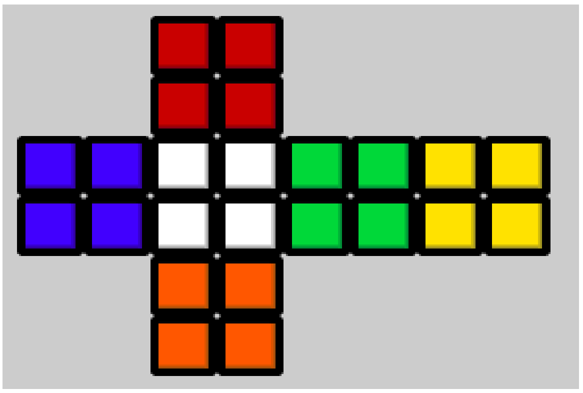


Illustration 18: 'dlbur' + 'DLBLD'

- Note that the agent only takes color information on the 6 faces as input, it's really interesting to see that it can recover these test cases

The modified Q-Learning Agent (with a neural network for function approximation)

The following table shows the final testing result after 3 train-test rounds for the modified Q-Learning Agent:

PL	1	2	3	4	5	6	7	8	9	10
RR	100	100	98	74	61	44	34	25	19	14
AM	1.00	1.67	2.66	7.22	9.47	12.45	14.26	15.89	16.83	17.62
PL	11	12	13	14	15	16	17	18	19	20
RR	11	8	6	5	4	3	2	2	1	1
AM	18.15	18.63	18.93	19.16	19.38	19.49	19.65	19.70	19.76	19.82

Table 2: final test result, modified Q-Learning, PL=permutation length, RR=recovery rate in percent, AM=average moves to recover

Below is the overall recovery rate across all permutation lengths in each round:

Round	1	2	3
Overall Recovery Rate	28.7%	29.7%	30.5%

Table 3: overall recovery rate, QNN

It can be seen that after 3 rounds of training (takes roughly 10 hours in my laptop), the QNN agent learns to solve around 61% of test cubes that are 5 steps permuted. Also the overall recovery rate in

9 https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php

each round does go up. This is very exciting result because the neural network only has 2 hidden layers with 19164 trainable parameters in total, and it manages to cover a fair amount of the state space. Compared with the Q-Learning agent, the neural network function approximation definitely has its advantage in terms of expressiveness and memory efficiency. Some manual inspections are done too, and the result is as follows:

Permutation	Solve Moves	Solved
F	b	True
d	U	True
Fu	Ub	True
dl	LU	True
LBd	Dbr	True
DDD	U	True
FRUBL	UUUUUUUUUU	False
dlbur	UBLUF	True

Table 4: manual inspection of QNN

Several points to note:

- For the first 6 tests, it's obvious that the agent mainly solves them by reversing the permutation process, and when the cube complexity is reduced to a simple state, it solves it by one face rotation that's not necessarily reversing the permutation (it solves ‘F’ with ‘b’ instead of ‘f’).
- Note that the agent only takes facelet color as input, it has no knowledge of what the permutation was.
- For the failed case, the agent doesn't seem to recognize the position, so it turns the UP face again and again hoping to result in a position that it knows how to solve, but unfortunately it fails to find one
- For the last test case, the agent successfully solves the cube, what's interesting is that **it doesn't solve by reversing the permutation process**, which should be ‘RUBLD’, instead, it chooses to solve with ‘UBLUF’. This means that the QNN agent not only learns to solve by "memorizing", but also discovered some alternative paths. More specifically, the agent solves this seemingly hard position to a good position that’s rotated from the original cube:

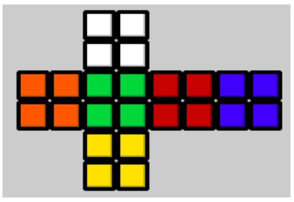


Illustration 19: original position

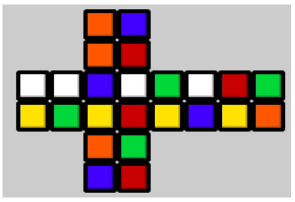


Illustration 20: "dlbur"

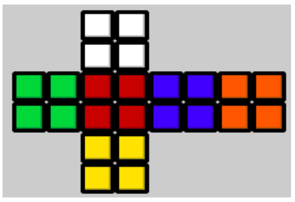


Illustration 21: "dlbur" + "UBLUF"

Justification

Compared with the earlier empirical number 5, the Q-Learning agent solves 5 permutation length positions with a 89% recovery rate, better than 61% of the QNN agent. It should be taken into consideration though, that since it takes very long to train the QNN model, I only trained the QNN agent 3 rounds versus 100 rounds of Train-Test for the Q-Learning agent. I think it’s safe to say that to some degree, both models achieve to “learn” to solve the pocket cube, at least for positions that are seemingly difficult for human (me).

An exciting result from these experiments is that the agents don’t simply “remember” the permutation, but also find alternative paths to solve the positions. To be exact, since they don’t know the permutation but only the colors of the faces as input, it’s expected that they come up with their “own” ways to achieve a “good” position. One thing to notice in the manual inspection parts is that both the Q-Learning agent and the QNN agent manage to solve “dlbur” but along different paths (“DLBLD” vs. “UBLUF”). This means that different algorithm, different model setup and even different training may result in different behaviors.

It must be made clear that the state space is very huge, and what both agents covered are just a fraction of it. A powerful search algorithm that enumerates all possible positions with allowed rotation moves can result in a much better agent (one that only “remembers” the permutation moves and reverse it). But the results of this project has sufficiently shown that the Q-Learning agents with several hours of training can beat me at these kind of problems:

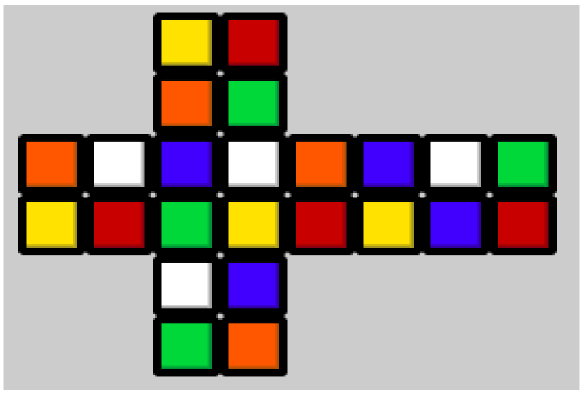


Illustration 22: "FRUBL"

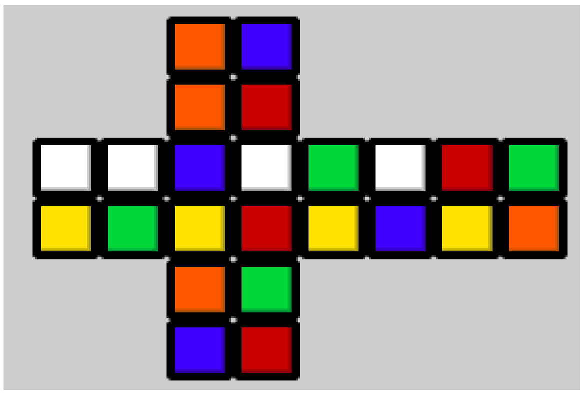


Illustration 23: dlbur

V. Conclusion

Free-Form Visualization

For this part, I figured there's no better way than just giving the audience a cube to solve. I have manually confirmed that the QNN has managed to solve this position. This way, you can get an idea of how powerful and stupid the model is.

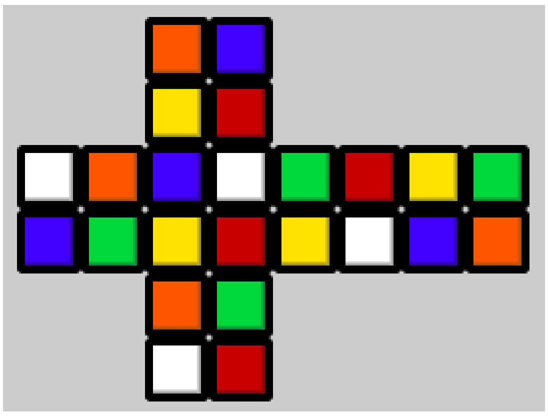


Illustration 24: fun for audience

Powerful because you'll feel it. Stupid because I had to manually try several permutation till I find one that the not-so-well-trained neural network knows how to solve. By the way, the permutation is "ldbfu".

Reflection

For this project, a pocket cube is first implemented to be the experiment environment. Then, a classic Q-Learning is trained and tested to solve the cube. Small specifications of the Q-Learning algorithm has been done, mainly:

Reward 100 for a good position, and none for other positions. The logic behind is that I don't want to give the agents any extra incentives on how to solve the cube (One could argue that rewarding 1 or 2 moves permuted cubes may speed up the training, but what about rewarding the agent when it follows a standard cube-solving algorithm like layer by layer, the boundary is unclear for me).

Setting alpha to be 1 and initialize at 0 Q-Values, which lead to the Q-Table clean up method. The reason is that, since I'm rewarding 100-0, I'm implicitly saying there are only "good" and "bad" positions for the agent. There's no point to propagate random values around the Q-Table, because in this alpha and reward setup, a large Q-Value from the next state propagating to the current state means nothing if the agent doesn't know how to solve the next state, which is initialized randomly. This way, the all 0 state entries in the Q-Table can be seen as visited positions by the agent, but the agent just doesn't know anything about the state. Hence, I can clean them up to save memory.

A modified Q-Learning agent with a neural network for function approximation is then trained and tested. I followed the "Deep Q-Learning" algorithm when implementing this. But you may noticed that I’ve been avoiding to use the “Deep” term, since my neural network only has 2 hidden layers. For this part of the project:

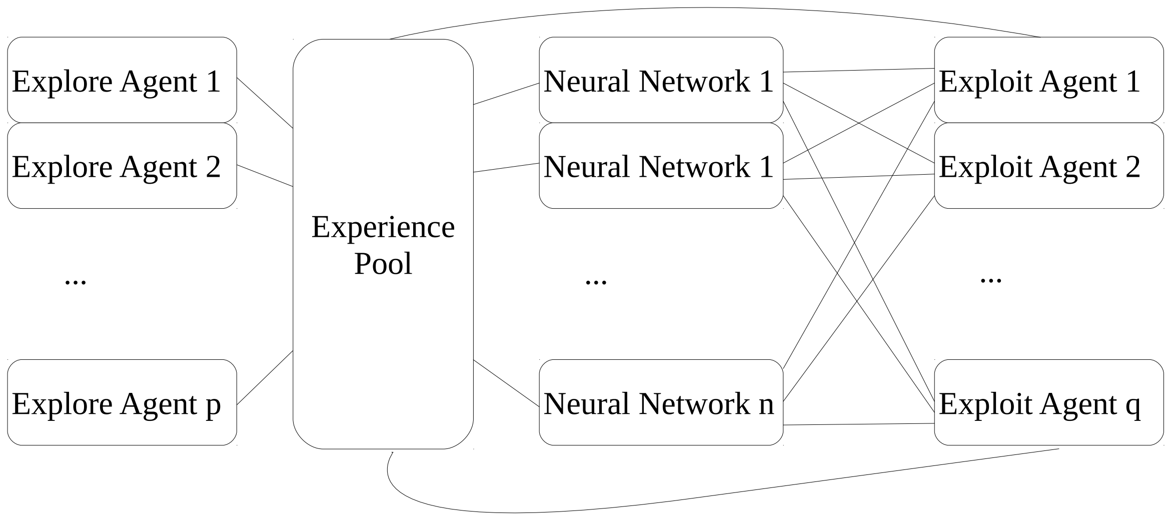
The convergence of the neural network is the key. Memory replay and batch size setup are all because of this. I also coded a piece of code that makes it possible to monitor the neural network’s output during training. In machine learning, overfitting is a term that’s often brought up. In a way, I should say my goal is to avoid overfitting and achieve overfitting at the same time. I try to avoid overfitting to a short training period, but I want the neural network to overfit to the Q-Table as much as possible.

The expressiveness of a neural network is truly amazing. With only 2 hidden layers and limited number of nodes, the QNN agent achieved a comparable result as what a Q-Learning agent with a huge Q-Table can do. However the training time for neural network is very frustrating too.

Comparing the two models, I guess one can say, it’s a trade-off between memory consumption and computing power. The Q-Learning agent is faster to train, but maintains a large Q-Table, the QNN model is expensive to train, but only maintains a small amount of parameters. Note that both models are fast in testing, one does a simple lookup in the Q-Table, the other does a simple forward propagation. What’s more, it’s a trade-off between discrete representation and continuous representation. For the cube solving problem, the state space and the action space are both discrete, a Q-Table can be learned, but when some infinite amount of state or action is relevant in the problem, the nonlinear expressiveness of a neural network is what makes it much more suitable.

Improvement

One problem that’s been haunting me throughout the project is the training time (or limited computing resources). For the Q-Learning agent I had to come up with a way to clean up the Q-Table during training, for the QNN agent I had to refrain with only 3 Train-Test rounds because that already takes a whole night on my poor laptop. It’s been shown in both models that the recovery rates are still going up near the end of the Train-Test rounds. It will be interesting to see what would happen if the QNN model is trained up until, say 1000 rounds. Will the 2 hidden layer structure be sufficient to fit even higher-order permutations? Or maybe more complex structures are necessary? These remain questions until the neural network is trained for longer time with more examples. Thus, I think a major improvement will be to parallelize the training part. In the QNN model, this is particularly relevant because with memory replay, the state-action-reward-nextstate information gathering and neural network fitting processes are asynchronous. I’ve not much experience with parallel computing, so here’s a naive structure that I’ve been thinking about:



There are p explore agents interacting with the environment, gathering state-action-reward-nextstate information to throw in the experience pool. There can be n different neural network structures that are trained and tested. There are p exploit agents that choose from the networks to act in the environment and also throw their experiences into the experience pool. There are several exciting points about this structure from my point of view:

- The process of explore agents acting in the environment, the process of training neural networks, the process exploit agents choosing neural networks and the process of exploit agents acting in the environment are all separated. This way, parallel implementation is possible.
- The structure fits well with the memory replay technique in Deep Q-Learning. The fitting of neural networks can be treated as standard classification/regression problems.
- The exploration-exploitation trade-off now translate to a p/q ratio. Changing the amount of agents interacting with the environment is a sufficient way of implementing the exploration-exploitation factor. A decay function can be embedded in a central control system too.
- This structure allows for training of multiple neural networks at the same time. This gives a more explainable way of choosing a neural network structure, or even… (next point)
- Borrowing the idea of genetic algorithms, why not let the nature of the problem choose the correct neural network structure, or even… (next point)
- Keeping all the neural network structures, and the interconnections between the neural networks and the exploit agents can be seen as a matrix of trainable weights, with some defined loss function, these weights can be optimized so that the exploit agents get the best expected outcome in the environment, or even… (next point)
- Borrowing the idea of ensemble learning, why not ensemble the exploit agents to form a “super” agent that ultimately becomes our solution to the problem, and let it finally engage in the real problem.

These are some of the intriguing ideas that came to me, for me, the possibilities that are opened up when expanding the already powerful Deep Q-Learning to a higher, more parallel and more complex level are truly amazing.

References

https://en.wikipedia.org/wiki/Rubik%27s_Cube

https://en.wikipedia.org/wiki/Pocket_Cube

https://en.wikipedia.org/wiki/Q-learning

https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

https://ruwix.com/online-puzzle-simulators/2x2x2-pocket-cube-simulator.php