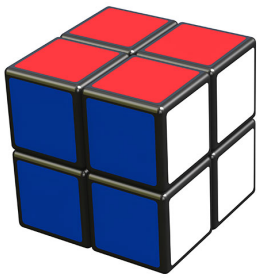
# Solving a Rubik's Cube using Deep Reinforcement Learning



Andreas Hagen, Fredrik Johanssen & Torstein Gombos

TEK5040

University of Oslo

UiO

November 30, 2018

# Abstract

This project aims to solve a scrambled 2x2x2 Rubik's cube, referred to as the pocket cube, using deep reinforcement learning. The main goal of this project is teaching an agent to find the correct rotation patterns for solving the cube.

The pocket cube has 3.674.160 different states, making it almost impossible to solve by random twisting. It is therefore essential to make a good training model in order to foresee the correct rotations during testing.

The algorithm shall learn and optimize actions based on corresponding rewards. Everything is programmed in Python 3.6, and the neural net is implemented with François Chollet's "Keras" module.

Solving the cube works successfully under certain conditions. Evaluating the algorithm, provides almost 100% accuracy up to 5 scrambles. From 6 to 15 scrambles, the accuracy rate is dropping down from 75% to 10%. A visualization of the accuracy plot is found under section 4.2.4.

# Contents


**1 Introduction** **5**

**2 Environment** **6**

2.1 The Rubik's cube . . . . . 6

2.2 Representation . . . . . 6

2.3 Action Space . . . . . 7

2.3.1 Scrambling the Cube . . . . . 7

2.3.2 Rendering the cube . . . . . 8

**3 Solving the Cube** **9**

3.1 Expectations . . . . . 9

3.2 Deep Q-learning . . . . . 10

3.2.1 Deep Q-Learning algorithm . . . . . 10

3.2.2 Exploration with $\epsilon$ greedy . . . . . 12

3.2.3 Training . . . . . 12

3.2.4 Batch creation . . . . . 12

3.3 Network Architecture . . . . . 13

3.3.1 Input Data . . . . . 13

3.3.2 Fully-Connected Feed Forward Neural Network . . . . . 14

3.3.3 Convolutional Neural Network . . . . . 14

3.3.4 Loss . . . . . 15

3.3.5 Optimizer . . . . . 16

3.4 Evaluating the network . . . . . 17

**4 Results** **18**

4.1 Evaluating the agent . . . . . 18

4.2 Fully connected feed forward network . . . . . 18

4.2.1 Agent 1 - No dropout or batch normalization . . . . . 18

4.2.2 Agent 2 - $\epsilon$ greedy . . . . . 20

4.2.3 Agent 3 - Directly at 5 scrambles . . . . . 21

4.2.4 Agent 4 - Fully connected with dropout rates . . . . . 22
4.3 Convolutional neural network . . . . . 24
4.3.1 Agent 5 - Convolutional Neural Network . . . . . 24

**5 Discussion** **26**
5.1 Generalization . . . . . 26
5.2 Replay modifications . . . . . 26
5.3 Reward . . . . . 27
5.4 Adaptive exploration . . . . . 28

**6 Conclusion** **29**

**References** **31**

**Appendices** **33**

**A Code** **33**
A.1 How to run . . . . . 33
A.2 Config.ini . . . . . 34
A.3 Environment.py . . . . . 35
A.4 Ruby_AI.py . . . . . 35

**B Documentation of code** **35**

# 1 Introduction

Reinforcement learning is based on the creation of an intelligent agent that learns and adapt to its environment. Its mission is to constantly improve the policy of actions by evaluating the existing solution (Solberg, 2018)(Simonini, 2018).

This report will focus on implementing an environment for the cube. Then train an agent to solve randomly scrambled cubes, using deep reinforcement learning. The goal is to test different networks in order to see how effectively an agent can be trained.

Already existing work on solving the Rubik's cube can be found in the article "Solving the Rubik's Cube Without Human Knowledge" (McAleer, Agostinelli, Shmakov, & Baldi, 2018). They have developed a capable agent they call *Deep Cube*. Other work include Jason Rute's GitHub repository (Rute, 2017) which has a good method of comparing the progress of the various networks. Both have provided this project with valuable knowledge and inspiration.

Figure 1 illustrates the process. Starting with a scrambled cube as the environment. The agent observes and predicts the best action. It then provides the user with a solved cube as an output. The reward for the action is used to train the network.

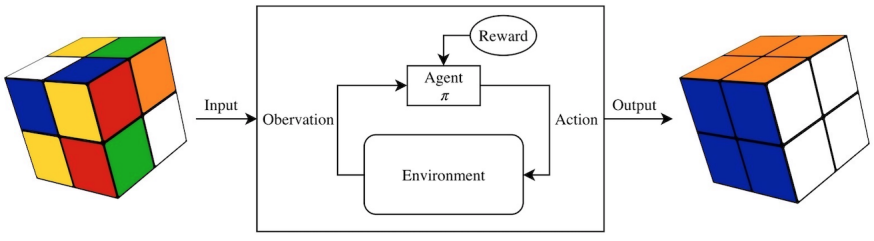



Figure 1: Solving the cube illustrated with reinforcement learning

# 2 Environment

This section aims to describe the environment, i.e. the Rubik's cube, and how it is represented in this project. It describes cube notation and corresponding actions. As well as describing the rules for scrambling the cube.

## 2.1 The Rubik's cube

In this project the pocket cube is chosen instead of the normal 3x3x3 Rubik's cube. The pocket cube is the two layered version of the popular Rubik's cube puzzle. Both invented by the Hungarian inventor, architect and professor Ernő Rubik (Ruwix, n.d.). The smaller version got 6 sides, called faces, consisting of 8 smaller corner cubes called cubelets. Each of these cubelets got 3 stickers attached with different colors. It has 24 stickers in total and 6 different colors.

There are a few reasons why the smaller pocket cube is used instead. It has a manageable amount of combinations compared to a normal Rubik's cube. This is a "proof of concept" project. If the agent is able to generalize a solution for the smaller cube, one can argue that it could be able to learn a normal cube. There is also a constraint on time and computational resources.

The pocket cube environment consist of 3.674.160 different states $\mathcal{S}$, whereas only one of the states, $s_{solved}$, is the solved state. Therefore solving by random it is near impossible to solve a fully scrambled cube.

## 2.2 Representation

The environment thinks of the cube as a 6x2x2 array. Each of the 6 elements represents the faces of the cube, and the 2x2 arrays are the stickers for that face. The stickers color is represented as an integer between 0 and 5. These values are One-Hot encoded, which is discussed in section 3.3.1, and used as input to

the deep learning model.

## 2.3 Action Space

The move method handles all rotations of the cube. It will always be initialized in a solved state. Moves are represented using Frey's and Singmaster's face notation, where a letter is stating which of the sides to turn (Frey & Singmaster, 1982). These letters corresponds to each of the 6 sides labeled up(U), left(L), front(F), right(R), back(B) and down(D). This project uses the quarter-turn metric, where one 90 degree turn, clockwise or counter-clockwise, counts as one move. Clockwise moves is denoted with a single character, e.g. U, and counter-clockwise moves is denoted with an $r$ (reverse) behind, e.g. Ur. Resulting in the action space: $\mathcal{A} = \{U, Ur, ..., D, Dr\}$, with 12 possible moves in total.

### 2.3.1 Scrambling the Cube

The environment contains a method for scrambling the cube. Where the cube needs to be scrambled in a specific manner.

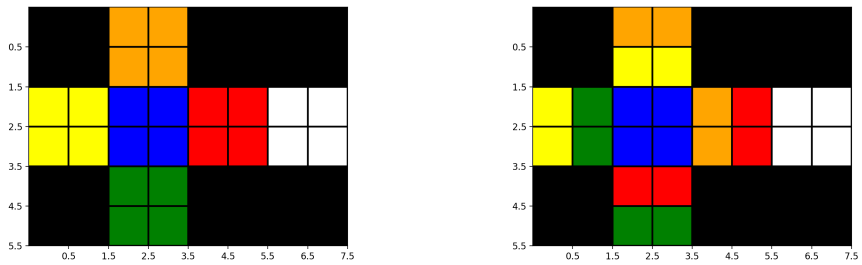
**Example:**

- Cube is scrambled 3 times
- Moves: F, L, then Lr.
- The environment has now nullified the last move. Same as only scrambling with the move F.

Training on this cube is like training on 1 scramble instead of 3. It is preferable that the agent trains on states it has not experienced before. Therefore, this has been taken into account. If the cube scrambled 4 times, it should take at least 4 moves to solve the cube. This feature is referred to as *Expanded Training*.

### 2.3.2 Rendering the cube

For visualization purposes a rendering option of the cube is implemented. This makes it easier to actually see what the network has learned and how it solves the cube. Figure 2 illustrates a render of the state before and after the front is turned clockwise.




(a) Solved state

(b) Next state

Figure 2: Rendering image of cube based on action. Face is rotated clockwise.

# 3 Solving the Cube

This section will cover how the agent trains to solve the pocket cube. A number of potential challenges are discussed in 3.1. The section will also explain concepts like deep Q-learning and $\epsilon - greedy$ which is used in the algorithm. Lastly, the architecture of the network and its evaluation is described.

## 3.1 Expectations

The agent should be able to generalize solutions for a cube, instead of brute forcing every possible state. In theory it should be possible to brute force a pocket cube, since it contains only 3.674.160 states. It would be preferable that it learns to generalize a solution.

To be able to find the best way of generalizing a solution, two different types of neural networks are tested. The first is a fully connected feed forward network, which uses the state of the cube reshaped as a vector as input. The second is a convolutional neural network, which uses the state of the cube directly as a 6x2x2 array.

An issue with solving a Rubik's cube using machine learning, is the large amount of possible combinations and sparse rewards. This makes it difficult to evaluate how close to a solution the cube is. To solve this, a *recursive reward system* is designed. This is done by evaluating the actions and rewards after each episode. If a policy solution ends with a solved cube then the program sets a reward to 1 for all the actions that led to the final reward. This ensures a correct label when calculating the error when updating the weights.

The agent should be able to solve the cube with as few moves as possible. To solve the cube with fewer moves, the number of allowed moves per cube is equal to the number of scrambles. The agent begins training the network with

cubes scrambled only once. Then increment the number of scrambles when it is deemed strong enough, described in section 3.4. The increase in difficulty is exponential due to the increase in the number of states. This will give the agent good knowledge about the states close to a solution, like 4 or 5 scrambles away. When the number of scrambles increases beyond that, the agent will ideally learn how to navigate back to familiar ground.

## 3.2 Deep Q-learning

Deep Q-Learning is a reinforcement algorithm. Standard Q-learning takes the current state and aims to maximize the reward for the actions taken from that state. This is refereed to as the Q-Values for that state (Brunskill, 2018). Equation 1 shows the optimization. Future rewards are taken into account as well, but with a discounted rate.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha((r_t + \gamma \underset{a'}{max}\, Q(s_{t+1}, a')) - Q(s_t, a_t)) \tag{1}$$

Deep Q-learning means that neural nets are used for the prediction of the Q-values. The state is the input of the network and the reward for the action taken is used as target. As illustrated in figure 3.

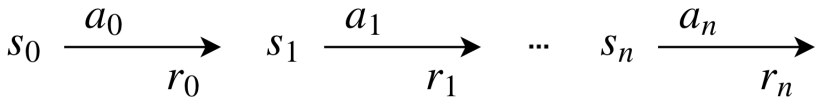



Figure 3: Relation between state, action and reward

### 3.2.1 Deep Q-Learning algorithm

The format for the learning follows the *State, Input, Reward and Next State* format. Often denoted as <S, A, R, S'>, demonstrated in algorithm 1. The

algorithm loops through episodes where it attempts to solve randomly scrambled cubes. When observing a state, $s_t \in \mathcal{S}$, at each time step, t, the agent predicts and perform an action, $a_t \in \mathcal{A}$, based on the state. After performing an action, the agent observes the new state of the cube, $s_{t+1}$. A reward is given based on the new state, 1 if it is the solved state and 0 otherwise. An episode ends, either by solving the cube or by exceeding the number of allowed moves. The program then saves the cube state, actions and rewards to memory. It will then train on this memory variable.

**Algorithm 1** SARS' algorithm

**while** *True* **do**
- **Scramble Cube n times**

**for** *Number of allowed moves* **do**
- **S** = Get state of cube
- **A** = Choose policy with $\epsilon - greedy$
- **R** = Check reward for performing action
- **S'** = Get the new state of the cube
- **if** *Cube is Solved* **then**
  - break
- **else**
  - Repeat until solved or exceeded maximum number of moves
- **end**

**end**
- **Add the <S, A, R, S> from the episode to memory**
- **Training**
- **Evaluate training**
- **Repeat**

**end**

### 3.2.2 Exploration with $\epsilon$ greedy

Choosing policy with $\epsilon - greedy$ makes the agent use exploration as well as exploitation. It starts with $\epsilon = 1$, which means that the agent focuses 100% on exploration. As the episodes goes on, exploration will decrease and exploitation will increase. When the agent evaluates that it is strong enough to handle a new scramble length, the exploration is reset to 1.

### 3.2.3 Training

The machine learning module used to train the agent is Tensorflow's Keras module. Keras is a high level machine learning API. It is used in combination with a Deep Q-Learning algorithm.

The agent will start training on single scrambled cubes. When it is able to handle its current amount of scrambles, the difficulty increases by adding another scramble. During episodes, experience of the state, action and rewards are stored in a memory variable. When an episode is over, the agent evaluates the episode with the method *go_to_gym()*. It samples a batch from the memory pool and trains on that sample. The input is the state and the target is the respective reward for the action that was chosen.

### 3.2.4 Batch creation

Often, the batch is randomly sampled after some distribution. However, when the amount of scrambles increases, it becomes important that training happens on a state it has not seen before. With a uniform random sampling, there would be a chance the agent never trains on the new set of cubes from the extra scramble length. To solve this the maximum memory size is set to the same size as the sample batch. When a batch is sampled it will train on all the current data available in the memory. It is shuffled to increase generalization.

## 3.3 Network Architecture

There are two possible network options. A convolutional neural network, and a fully-connected neural network. The main difference is the input to each network. The fully connected network reshapes the cube state to a 1-dimensional vector. The convolutional network uses the cube state as 3-dimensional array. Both inputs are treated with One-Hot encoding.

### 3.3.1 Input Data

The current state of the cube is used as input data to the two networks. It is reshaped from 6x2x2 to 1x144 for the fully-connected network, and 1x6x2x12 for the convolutional network. The output represents the action space and is shaped as a 1x12 array. The input is seen as *categorical data*, and the actual value between 0 and 5 do not matter. Therefore, One-Hot encoding is implemented. One-Hot encoding is a way to handle categorical data so it is easier for the network to understand (Warakagoda, 2018). The operation converts all the possible categories into binary vectors. This means that a sticker with the value 0 would be encoded to [0, 0, 0, 0, 0, 1], and a sticker with the value 3 would be [0, 0, 0, 1, 0, 0]. Table 1 illustrates the encoding.

| White | Blue | Orange | Red | Green | Yellow |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 |

Table 1: The network's One-Hot encoded stickers.

### 3.3.2 Fully-Connected Feed Forward Neural Network

The design of the network is based on *Deep Cube* architecture (McAleer et al., 2018). It consists of 4 dense layers. Where the first three uses the *rectified linear unit activation function* (ReLU). The last one defines the number of outputs and uses the *softmax* activation function. This activation functions returns the probabilities for choosing each action. Representing the expected reward for the agent. Each layer uses a dropout of 20% to increase generalization and reduce chance of overfitting. Figure 4 shows the network setup and units used.

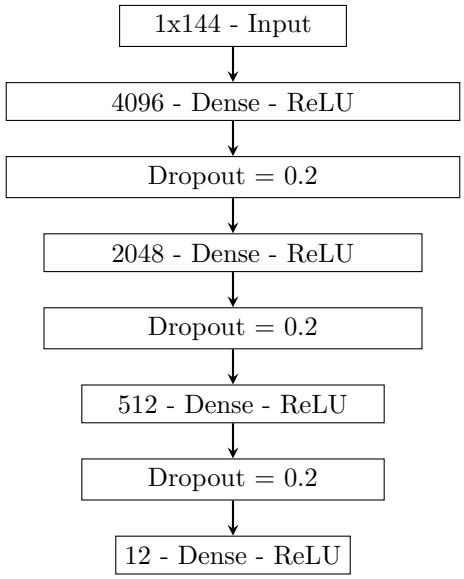



Figure 4: Architecture for the fully connected convolutional model

### 3.3.3 Convolutional Neural Network

This design is based on the principle of feature detection and classification (Khaksar, 2018). The first layer is a 2D convolutional layer with 256 filters. The second is the same, but with 128 filters. Both layers have kernel size = (2, 2), strides = (3, 3) and ReLU activation. The output of the second layer

is further flattened, and fed through two fully connected layers. Both of these have ReLU activation. The input is the One-Hot encoded state of the cube. Figure 5 shows the network setup. The output, loss and the optimizer is the same as for the fully-connected network.

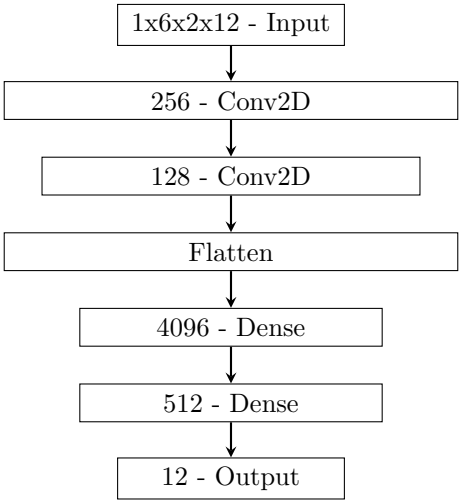



Figure 5: Architecture for the convolutional model

#### 3.3.4 Loss

The Softmax activation function returns the probability for choosing that state. This becomes the expected reward for a state after training. Since the output is between 0 and 1, it is natural to use a cross-entropy loss function. This loss function increases as the predicted value diverges from the target (*Loss Functions*, 2017), which is illustrated in 6.

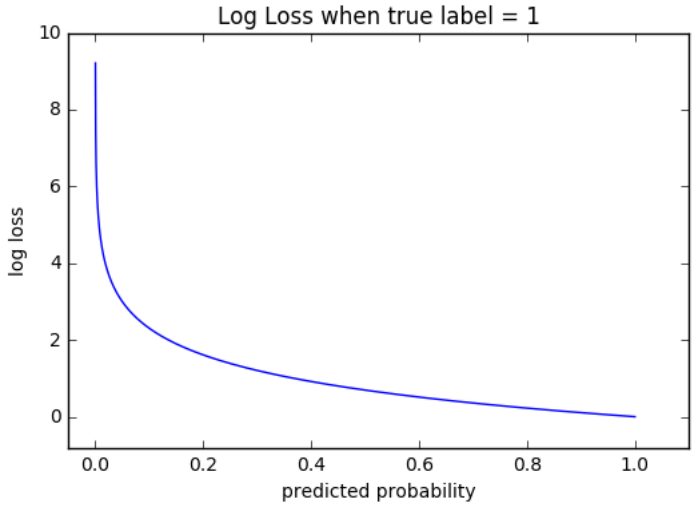



Figure 6: Cross entropy

### 3.3.5 Optimizer

The model uses the *Adadelta* optimizer. It is a gradient descent optimizer with adaptable learning rate. Adadelta's learning rate is calculated recursively from the decaying of the average of the past gradients (Ruder, 2018). The *Deep Cube* project uses a similar optimizer called *RMSprop*. Though through testing, AdaDelta provided the most promising results.

The optimizer has little momentum, which is a feature used to escape local optima (Ellefsen, 2018). It starts with a high learning rate, and is considered more robust than *Adaboost* (csn231, n.d.). An initial learning rate of 1 is used, recommended by Keras' documentation on *Adadelta*.

## 3.4 Evaluating the network

Every 100'th cube, the agent checks the accuracy of how many were solved. If all 100 are solved, the agents proceeds to evaluate the network. The evaluation method will try to solve 1000 cubes with its current network. If evaluation accuracy is over 97%, the network is deemed strong enough. The number of scrambles then increases by 1. If not it will return to training. This evaluation happens every time it solves 100 cubes during training. Figure 7 demonstrates this.

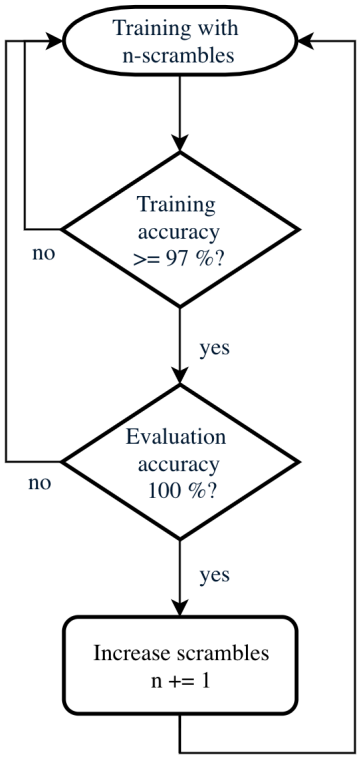



Figure 7: The training program. If evaluation accuracy is over 97%, difficulty increases by one scramble

# 4 Results

This section presents the results from the various tests and methods implemented. Several agents were trained under different conditions. First describing the training of an agent, then presenting the results from testing that agent.

## 4.1 Evaluating the agent

The test of the agent is done by starting at 1 scramble and increasing the scramble length until it reaches 15 scrambles. For each scramble length, the agent evaluates how well it is able to solve 1000 cubes.

## 4.2 Fully connected feed forward network

Four agents have been tested with a fully connected feed forward network:

### 4.2.1 Agent 1 - No dropout or batch normalization

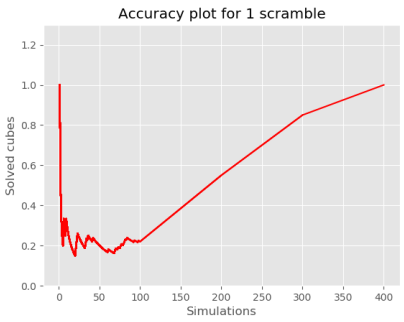
**Training**

| Dropout | Batch norm. | Policy | Expanded training | N cubes |
|---|---|---|---|---|
| No | No | greedy | No | 5 000 000 |

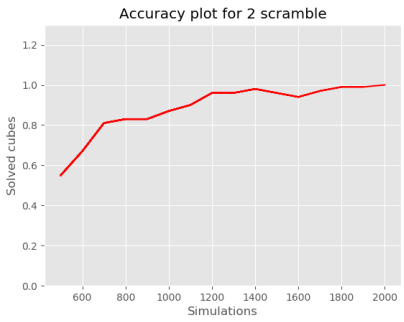
Table 2: Agent 1

As seen in table 2 the agent was trained without dropout and batch normalization. It follows a greedy policy, meaning there was no exploration. It is not using the *expand training* feature, therefore the agent trained on some states it had seen before. During training it looked at approximate 5 000 000 cubes. The agent had a fast learning curve. By about 500 000 cubes, 6 scrambles was

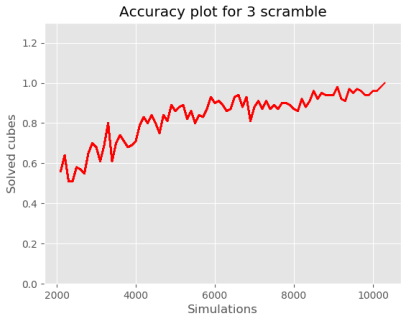
achieved. Here it first seemed to get stuck in local optima with around 50%-60% accuracy. Then the accuracy started dropping. After a few millions it had an accuracy of around 5-10%. Figure 8 shows the learning speed up to 3 scrambles.




(a) Training accuracy for 1 scramble




(b) Training accuracy for 2 scrambles




(c) Training accuracy for 3 scrambles

Figure 8: The learning progress from 1 to 3 scrambles

### Testing

As seen in figure 9, the network is able to solve the first 3 scrambles with almost 100% accuracy. After 3 scrambles the accuracy quickly converge to 0%. When taking a closer look at what actions the agent do, it seems like it gets stuck in some local optimum. Looking at the rendering of the states, it is close to a solution, but unable to solve it. The agent is twisting in and out of the same state until there are no more available moves.

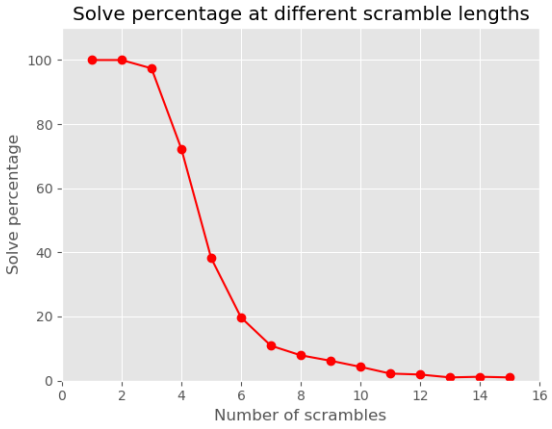



Figure 9: Results from testing agent 1.

### 4.2.2 Agent 2 - $\epsilon$ greedy

**Training**

| Dropout | Batch norm. | Policy | Expanded training | n cubes |
|---|---|---|---|---|
| Yes | Yes | $\epsilon$ - greedy | Yes | 1 000 000 |

Table 3: Agent 2

Results of testing the first agent, revealed an agent stuck in several local optimum. To account for this behaviour an agent which implements the $\epsilon$ - greedy policy was trained. It starts with a high exploration rate and slowly exchanges exploration with exploitation. Minimum exploration rate was set to 0.01. This means that every 100'th step a random move is chosen. The evaluation threshold was lowered to account for the exploration. The agent looked at approximately 1 000 000 cubes. 4 scrambles was reached, but again the accuracy goes down to almost 0. Training was then ended.

#### Testing

As seen in figure 10, the agent was only able to solve 1 scramble with 25% accuracy. It converges to 0% at 4 scrambles, meaning this agent did worse than the first agent.

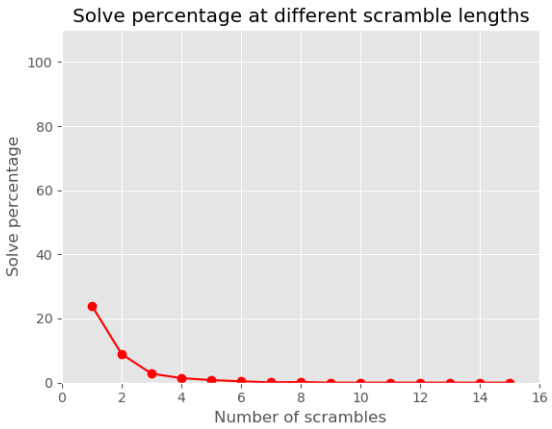



Figure 10: Results from testing agent 2.

### 4.2.3 Agent 3 - Directly at 5 scrambles

#### Training

| Dropout | Batch norm. | Policy | Expanded training | n cubes |
|---|---|---|---|---|
| Yes | Yes | greedy | Yes | 1 000 000 |

Table 4: Agent 3

Started training an agent directly at 5 scrambles. This was to check if training iterative from 1 scramble length was the optimal choice. Perhaps a larger number of moves and higher difficulty would train an agent to generalize a solution better.

This agents maximum number of moves was set to 10 instead of scramble length. During training, the maximum accuracy achieved was 8%. This only happened once, and the average was about 2%

**Testing**

This approach resulted in an accuracy of approximate 70% at 1 scramble, as seen in figure 11. It quickly converged to 0% and is quite similar to results of the $\epsilon$ - greedy agent.

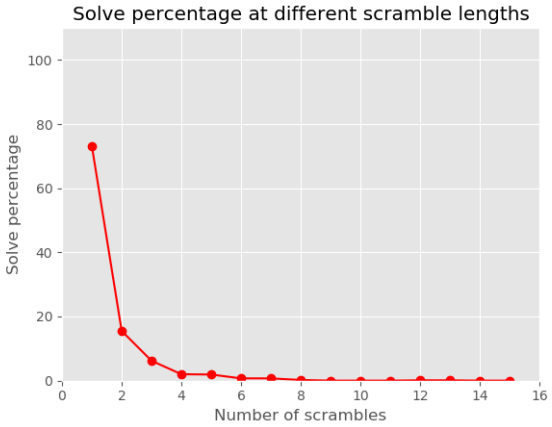



Figure 11: Results from testing agent 3.

### 4.2.4 Agent 4 - Fully connected with dropout rates

**Training**

| Dropout | Batch norm. | Policy | Expanded training | n cubes |
|---|---|---|---|---|
| No | No | greedy | No | 5 000 000 |

Table 5: Agent 4

This agent used 20% dropout for each layer. Dropout was used to avoid an

overfitted net and to force the agent to generalize solutions for the cubes.

The training learned quickly and reached 6 scrambles after about 250 000 cubes. It seemed to have learn some sort of generalized solution, because it managed to solve about 60% right after it started on 6 scrambles. The accuracy increased to about 60-80%, but with a lot of variance. Then the accuracy started dropping. After about 5 000 000 cubes, the accuracy was around 20% and the training was stopped.

**Testing**

When testing this agent, the network was at its peak performance, training on 6 scrambles. Almost 100% accuracy within the first 5 scrambles. 6 scrambles manages 75%. The accuracy drops steady, but as seen in figure 12, it is able to solve 15 scrambles with about 9-10%.

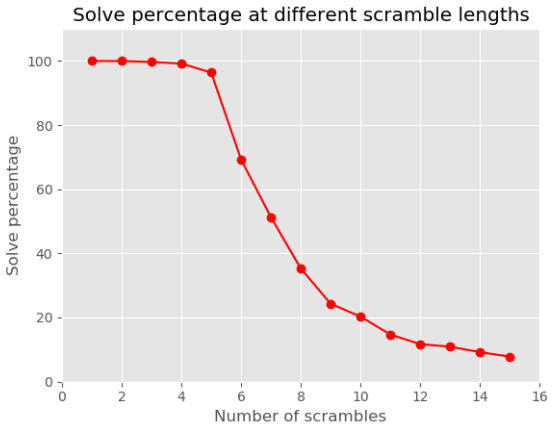



Figure 12: Results from testing agent 4.

## 4.3 Convolutional neural network

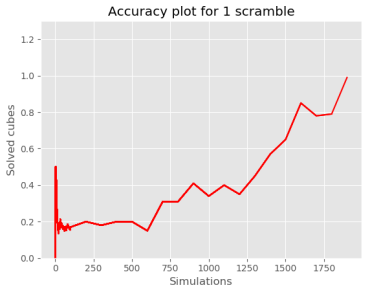
### 4.3.1 Agent 5 - Convolutional Neural Network

**Training**

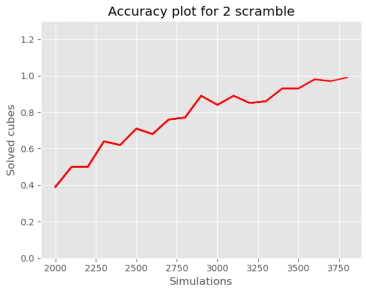
| Dropout | Batch norm. | Policy | Expanded training | n cubes |
|---|---|---|---|---|
| No | No | greedy | No | 5 000 000 |

Table 6: Agent 5

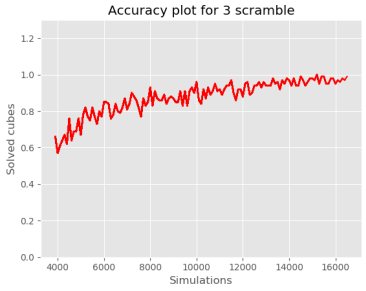
This agent was trained in the same manner as agent 1. The agent learned slower than agent 1 and only reached 4 scrambles.



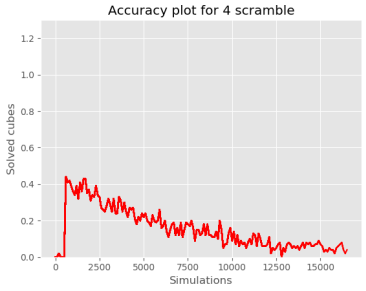

(a) Training accuracy for 1 scramble




(b) Training accuracy for 2 scrambles




(c) Training accuracy for 3 scrambles




(d) Training accuracy for 4 scrambles

Figure 13: The learning progress from 1 to 4 scrambles

### Testing

Even though the training performed poorly, there was hope that i would manage a generalized solution for a high amount of scrambles. For this particular agent, this did not seem to be the case. It quickly drops down almost no accuracy for the higher scrambles.

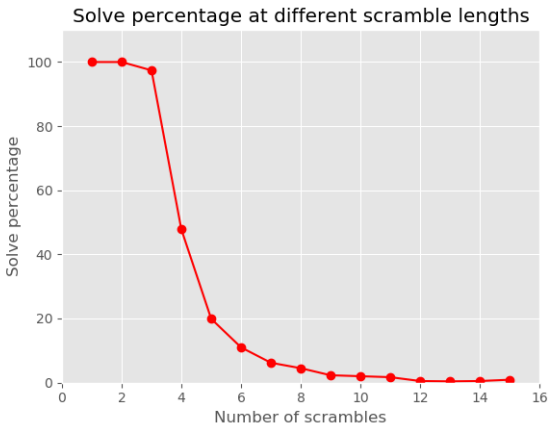



Figure 14: Results from testing agent 5.

# 5 Discussion

This section discusses the results and possible future improvements to the methods.

## 5.1 Generalization

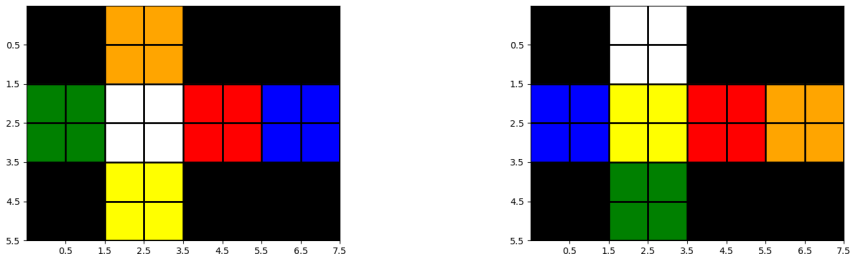
Seen from the results, many of the trained agents drops in accuracy when the number of possible states becomes too large. It looks like the network struggles to generalize a solution and instead tries to learn every possible state. Some solutions seems generalized, since it is able to solve a handful of cubes with a high number of scrambles.

A possible solution to increase generalization is to randomize the colors of the cube faces each time it resets. For example, the white side of the cube might switch with the red side of the cube. Resulting in a reduced number of states the agent would have to learn. If the cubes in figure 15 was scrambled with the same moves, they would still be solved with the same policy of actions. Even though they would look completely different to the agent. The network could benefit from using convolutional layers in the beginning to extract these features better when the sides are randomized.

A feature like the randomly reset cube is implemented in the code, but there was not enough time to test it properly.

## 5.2 Replay modifications

When the network reaches a certain amount of scrambles, testing reveals that it is able to handle less scramble lengths with almost 100% accuracy. It would be unnecessary to train on states the agent already knows. A possible solutions is to force the agent to train on scrambles it has not seen before. For an episode




(a) Random generated cube 1

(b) Random generated cube 2

Figure 15: Different cubes with the same scrambles will also have the same solutions

with e.g. 5 scrambles, only save the beginning of the episode. This would mostly consist of states it has not been exposed to before. It should probably still look at solutions close to a solved cube. Though this could perhaps only occur with a certain probability. This would reduce the training time when the number of scrambles increases

## 5.3 Reward

The way the reward system is implemented in this project works to some degree. Though the lack of generalization suggests that it is not the most ideal choice.

In a video from a "PyData" meet-up, software engineer Szymon Matejczyk discusses some of the challenges using machine learning to solve this problem (Matejczyk, 2017). Claiming, *how* one defines reward, is the most important factor. Many of the other techniques will simply define how fast the network learns.

When a cube is several moves away, it could be possible to set minor rewards when reaching milestones. Like a reward for percentage of the cube solved. Or a reward for reaching checkpoints in a human made algorithm. For any of these

methods to have a chance of success, a high exploration rate is needed. Without that, it would most likely converge on a local optimum. It would also need a lot more moves per cube, So the agent has room to solve it.

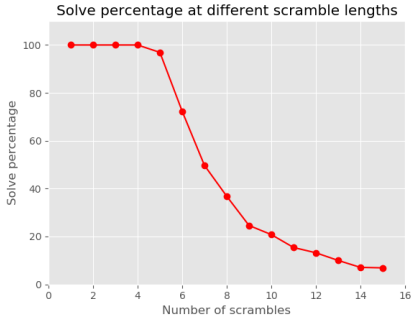
A possible issue with the environment ways of handling reward is that if the cube is not solved, no reward is given at all. The agent could reward itself for actions that previously led to rewards as well. And simply be content with redoing the same moves in a cycle. Rendering the cube when testing, reveals patterns that suggests this.

## 5.4 Adaptive exploration

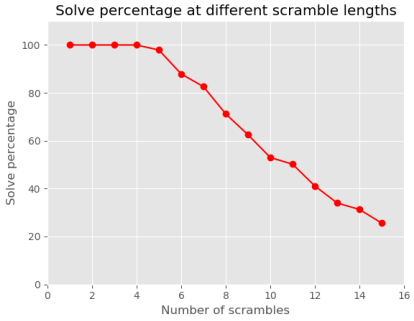
When a repetitive cycle is occurring exploration is important to break out of the local optima. A possible solution could be to adapt the exploration based on how much the accuracy is increasing or decreasing. If an agent seems to be stuck on a number of scrambles with little change in average accuracy, exploration should be increased. Increasing exploration should also increase the number of moves. This way, the agent can have room to see if the exploration pays off.

# 6 Conclusion

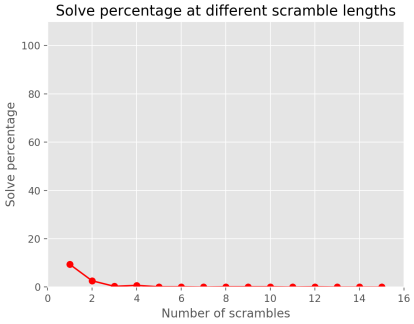
The main goal of this project was to teach an agent to find the correct rotation patterns for solving the cube. The agent is able to solve most cubes up to 6 scramble lengths, with the same amount of moves or less. This is better performance than humans with no knowledge of solving the cube. The results suggests that the agent memorizes how to solve each state seen instead of developing an algorithm to solve cubes in general.




(a) Results of the agent when evaluated on scrambles with the expand training feature.




(b) Evaluating the agent on unbiased scrambles, meaning no expand training.




(c) Evaluating an untrained agent with expanded training feature

Figure 16: Results from the best agent vs. an untrained agent

As shown in figure 16a, the best trained agent performs well with almost 100%

accuracy up to 5 scrambles. 6 scrambles drops to about 75%. 15 scrambles are solved with about 9-10%. Compared to the untrained agent in figure 16c, the agent clearly shows that it learns. The conclusion is that the agent is better than an untrained agent which chooses its actions randomly. Figure 16b shows the agent solving cubes that has no bias during the scrambling. This means that the scrambler do not care if it scrambles backwards. This is not how performance is evaluated in the results, but the World Cube Association do not to have any rules against this type of scramble sequences (WCA, n.d.). Which is why it is included in the conclusion. Under these rules, it naturally performs better, with almost linear drop in accuracy.

The lack of generalization is discussed in section 5. The biggest issues is the sparse rewards and the huge amount of states. The success the agent shows with scramble length 6, shows that it is possible to train an agent to solve a Rubik's cube. Given a more guiding reward system, or better implemented training as discussed in section 5, much better results could be achieved.

# References


Brunskill, E. (2018). *Model free control.* Retrieved from http://web.stanford.edu/class/cs234/slides/cs234_2018_l4.pdf

csn231. (n.d.). *Learning.* Retrieved from http://cs231n.github.io/neural-networks-3/

Ellefsen, K. O. (2018, September). *Multi-layer neural networks, biologically inspired computing.* Retrieved from https://www.uio.no/studier/emner/matnat/ifi/INF3490/h18/timeplan/slides/lecture6-6pp.pdf

Frey, A., & Singmaster, D. (1982). *Handbook of cubik math.* Enslow publisher.

Khaksar, W. (2018, October). *Deep learning.* Retrieved from https://www.uio.no/studier/emner/matnat/ifi/INF3490/h18/timeplan/slides/lecture-7-dl%281pp%29.pdf

*Loss functions.* (2017). Retrieved from https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

Matejczyk, S. (2017, November). *Learning to solve rubik's cube.* Retrieved from https://www.youtube.com/watch?v=5Mm6NQXVLAg

McAleer, S., Agostinelli, F., Shmakov, A., & Baldi, P. (2018). Solving the rubik's cube without human knowledge. *https://arxiv.org/abs/1805.07470*.

Ruder, S. (2018). *An overview of gradient descent optimization algorithms.* Retrieved from http://ruder.io/optimizing-gradient-descent/index.html#adadelta

Rute, J. (2017). *Solving the rubik's cube with deep reinforcement learning and monte carlo tree search.* Retrieved from https://github.com/jasonrute/puzzle_cube

Ruwix. (n.d.). *The 2x2x2 rubik's cube - beginner's solution.* Retrieved from https://ruwix.com/twisty-puzzles/2x2x2-rubiks-cube-pocket/

Simonini, T. (2018). *An introduction to reinforcement learning.* Retrieved from https://medium.freecodecamp.org/an-introduction-to-reinforcement-learning-4339519de419

Solberg, E. (2018). *Reinforcement learning.* Retrieved from https://www.uio.no/studier/emner/matnat/its/TEK5040/h18/lecture-slides/reinforcement-learning.pdf

Warakagoda, N. (2018). *Naive way of word representation.* Retrieved from https://www.uio.no/studier/emner/matnat/its/TEK5040/h18/lecture-slides/text-processing-deep.pdf

WCA. (n.d.). *Scrambling.* Retrieved from https://www.worldcubeassociation.org/regulations/#article-4-scrambling

# Appendices

## A Code

This section will give a brief explanation of the code and how to run it. There is three python scripts and one config file.

### A.1 How to run

**To start training:**

- Go to the config.ini file
- Change the train parameter to True and test parameter to False.
- Run Ruby_AI.py

This starts training a fully connected network to solve a pocket cube.

**To test a network:**

- Go to the config.ini file
- Change test parameter to True and train parameter to False.
- Set the path to the network for the *load_model* parameter.
- Run Ruby_AI.py

This will load a network and test it. It will start with one scramble and work itself up to 15 scrambles.

## A.2 Config.ini

The config file consist of three categories, where different parameters can be chosen.

- simulation
- network
- environment

In the simulation category it is possible to choose between training and testing, by setting it to true or false. If testing is set true, one have to load network weights under network category. Weights are saved in the models folder as a h5 file. It is possible to set *show plot* to true, where it shows a dynamic live plot of the accuracy at each scramble stage. The last variable, *difficulty level test*, is how many scrambles lengths the network should be tested on. E.g. 15, the network test 1000 cubes at each scramble length from 1 to 15, and displays a accuracy plot at the end.

Under the network category one can choose between fully connected and convolutional network. Simply by uncommenting the network one want to run. It is possible to choose parameters such as *batch size*, *memory size*, *epsilon values*, *learning rate values*, *epoch* and *threshold*. Where the *threshold* is the evaluation accuracy threshold, currently set to 97%. In this category it is also where one load the network weights, by inserting the name of a file in the models folder. Lastly, this is also where the *One-Hot encoding* is enabled, by setting it to true.

The last category, environment, have some few possibilities. One of them is *render image*, by setting it to true it renders a plot representing the cube. It renders one plot after each action, and by closing the plot it renders a new plot. This is only for visualization purposes. It is also possible to choose between numeric representation of the cube or not. At last, it is possible choose between

random sides and extended training or not.

### A.3 Environment.py

This is where the the representation of the cube is. It has the functions as seen in the documentation under Appendix B. When running this code it will simply render a image of the cube taking one action.

### A.4 Ruby_AI.py

This is the main program, where the different methods discussed in the project is implemented. The program consist of one class, *Network*, where all methods are implemented. The attributes and methods are listed in the documentation html for Ruby_AI. The script can be considered the hub for the agent. It interacts with the environment and trains itself by this interaction. The two main methods are the *train* and *test*. Train start by initiating episodes. Every episode starts resetting a cube and then scramble it. Afterwards it evaluates the episode and adds it to memory. After this, the network trains on episodes it has stored in memory.

## B Documentation of code

This part of the appendix includes the documentation of the python code, using pydoc. The documentation gives a brief explanation of all functions and the corresponding parameters.

# environment

index
c:\users\toanb\documents\skole_programmering\tek5040 prosjekt\environment.py

This is the environment script for the agent.

## Modules

configparser numpy random

## Classes

builtins.object

Cube

class **Cube**(builtins.object)

```
Class for representing the Rubik's Cube

Attributes:
    cube_size: The integer will represent a quadratic number. If cube_size = 2, then the cube is a 6x2x2
    cube: 6x2x2 array containing all the faces
    face: Dict with all the faces stored as arrays
```

Methods defined here:

**__init__**(self, cube_size=(6, 2, 2))
```
:param cube_size: Cube size is 2x2 by default
```

**__repr__**(self) -> str
```
Prints a string representing the array cube
:return:
```

**expand_training**(self, store_action, a)
```
Makes sure that the cube does not scramble backwards.
Will return an action that always scrambles a cube outwards.
:param store_action:
:param a:
:return:
```

**reset_cube**(self)
```
Sets a cube array to start position
:return:
```

**reward**(self)
```
Calculates the reward for the cube.
1 - Cube is Solved
0 - Cube is not solved
:return:
```

**rotate_cube**(self, face, render_image=False)
```
Move the faces of the cube in either clock-wise or counter-clock-wise
Movement is hardcoded to work on 2x2x6 cubes
:param face: The face to rotate
:param render_image: If True, render an image before and after the move
:return:
```

**scramble_cube**(self, k:int, render_image=False)
```
Takes in a cube array, and scramble it k times.
Returns the scrambled cube
:param render_image:
:param k:
:return:
```

---

Data descriptors defined here:

**__dict__**
```
dictionary for instance variables (if defined)
```

**__weakref__**
```
list of weak references to the object (if defined)
```

# Ruby_AI

index
c:\users\toanb\documents\skole_programmering\tek5040 prosjekt\ruby_ai.py

## Modules

| | | | |
|---|---|---|---|
| configparser | math | random | tensorflow |
| copy | numpy | re | time |
| keras | matplotlib.pyplot | sys | |

## Classes

builtins.object

Network

class **Network**(builtins.object)

```
Follow the algorithm State, Action, Reward, State'


Attributes:
    train_model
    test_model
    input_shape
    choose_net
    eta
    gamma
    decay
    epsilon
    epsilon_min
    epsilon_decay
    batch_size
    pretraining
    epoch
    threshold
    one_hot
    evaluate
    number_of_cubes_to_solve
    self.done
    self.solved
    self.epsilon_decay_steps
    self.simulations_this_scrambles

    self.memory

    self.difficulty_level
    self.best_accuracy
    self.difficulty_counter
    self.accuracy_points
    self.axis
    self.plot_progress
    self.check_level
    self.difficulty_test
    self.number_of_moves_eval


We also need to see how the autodidactic algorithm works into this, since the number of states
is about 3.6 million, it might use a really long time to train on this.
```

Methods defined here:

**__init__**(self, cube)

```
Initialize self.  See help(type(self)) for accurate signature.
```

**add_to_memory**(self, memory_temp)

```
Adds memory_tempt to memory and corrects the target vector.
:param memory_temp:
:return:
```

**check_progress**(self, simulation, solved_rate)

```
Checks the progress of the training and increases the number of scrambles if it deems
the network good enough
:param simulation:
:param solved_rate:
:return:
```

**create_target_vector**(self, next_state, reward, actions, take_action, end_reward)

Creates the target vector used as label in the training.
Will also look at future rewards with a discount factor
:param end_reward:
:param take_action:
:param next_state:
:param reward:
:param actions:
:return:

**evaluate_network**(self, cube)
Evaluates how well the network is performing over all.
:return:

**get_epsilon**(self, decay_steps)
Returns the e - greedy based on how many simulations that has been run for this
amount of scrambles
:param decay_steps: The number of simulations used for the current amount of scrambles.
:return:

**go_to_gym**(self)
Method that trains the network with batches of data
:return:

**load_network**(self)
Loads a pretrained network
:return:

**model_conv**(self)
Creates a convolutional neural network.
Input shape is 6x2x2x12 array and returns.
Returns the expected reward for a set of actions.
:return:

**model_fcn**(self)
Creates a Fully Connected neural network
Input shape is 1x144
Returns the expected reward for a set of actions.
:return:

**plot_accuracy**(self, simulation, solved, solved_rate)
Plots the accuracy of the cubes
:return:

**sample_memory**(self)
Creates a mini batch from the memory. Samples uniformly
:return:

**test**(self, cube, simulations=1000)
Method for testing a trained model
:param cube: The environment
:param simulations: Number of simulations
:return:

**train**(self, cube)
Trains the data with reinforcement learning.
Loops through episodes and stores experience to memory
:param cube:
:return:

---

Data descriptors defined here:

**__dict__**
dictionary for instance variables (if defined)

**__weakref__**
list of weak references to the object (if defined)

## Functions

**main**()
```
Main function of the script
:return:
```

**run_network_test**(model, rubiks_cube)
```
Runs an evaluation of a previously trained model
:param model:
:param rubiks_cube:
:return:
```

## Data

**config** = <configparser.ConfigParser object>