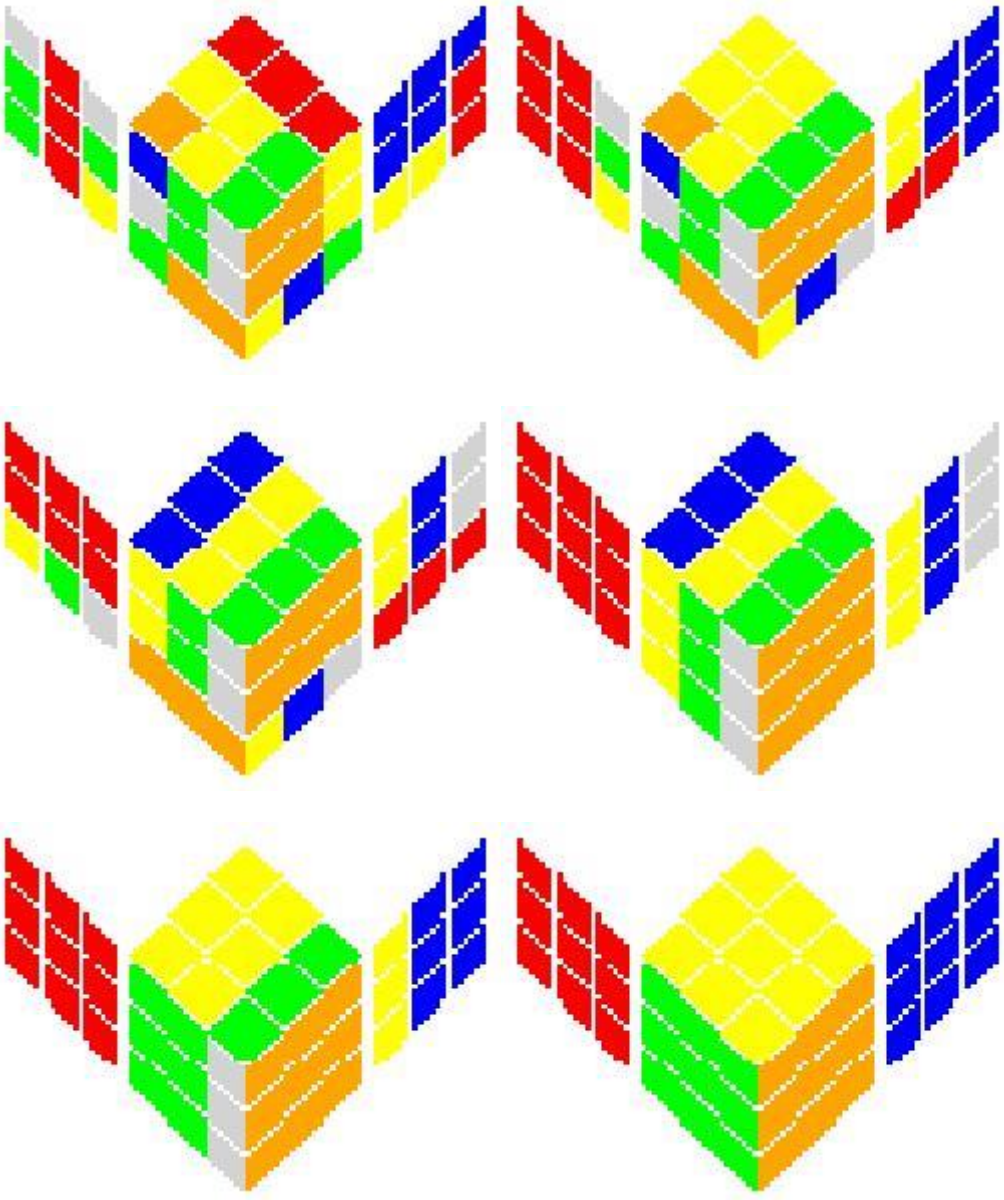
基于 DQN 的魔方还原游戏

1. 概述

魔方是一个看似简单实际状态极其复杂的游戏。我们基于 pypi 已有的 pycuber 模块自行编写了三阶魔方的 gym 环境。通过 DQN 网络（部分结构有修改），使用逐步训练的特殊方式训练游戏。

最终结果虽然不甚理想，但网络对于随机打乱 6 步及以下的魔方已经具备一定的还原能力。

下图给出一个例子：



还原公式：['B', 'L', 'D', "L'", "R']

2. 魔方环境

对应于 CuberEnv.py（image_map.py 和 cuber_img.py 用于魔方渲染和图像生成）

程序文档如下

```
'''
Rubik's Cuber:
     U         Upward
    LFRB  Left  Front  Right  Behind
     D         Downward

Goal:
    Given a picture of all side's blocks as initial state.
    Solve the cuber, with blocks of the same color in each
side.
    If step >= max_step without solved, game failed.

State(6 x 3 x 3):
    3x3(solved):
       000
       000
       000
    111222333444
    111222333444
    111222333444
       555
       555
       555

Action(12):
    F: Front clockwise
    F': Front anticlockwise
    L: Left clockwise
    L': Left anticlockwise
    R: Right clockwise
    R': Right anticlockwise
    U: Upside clockwise
    U': Upside anticlockwise
    D: Downside clockwise
    D': Downside anticlockwise
```

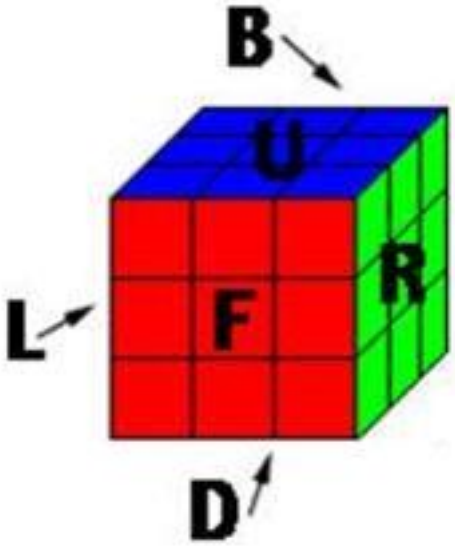
```
    B: Behind clockwise
    B': Behind anticlockwise

Reward:
    For each step:
        reward = -1
    If visited state:
        reward = -2
'''
```

详细说明如下

魔方的构成和表示：

U：上面 L：左面 F：前面 R：右面 B：后面 D：下面



游戏目标:

给定一幅包含魔方所有面（或 5 个面，由约束条件，另一个面唯一）的图片作为初始状态，解这个魔方，使其每个面的颜色均相同。如果步数大于设定的步数而魔方还未解完，则认为失败（如果基于贪婪策略，则如果相邻 2 步互逆或连续 4 步相同，则游戏陷入死循环，提前判定为失败）。

魔方状态的数组表示（6×3×3）：

一个已解好的魔方状态为：

```
                0  0  0
                0  0  0
                0  0  0
1  1  1  2  2  2  3  3  3  4  4  4
1  1  1  2  2  2  3  3  3  4  4  4
1  1  1  2  2  2  3  3  3  4  4  4
                5  5  5
                5  5  5
                5  5  5
```

魔方状态的图像表示（3 通道 92x76，渲染时大小为 256x192）：

注：我们在环境文件中编写了函数，可以通过魔方的五个面来推理最后一个面，但 DQN 网络训练仅输入五面的图像。

魔方的动作空间（数字索引将被转化为如下公式表示）：

F：前面顺时针旋转 F’：前面逆时针旋转

L：左面顺时针旋转 L’：左面逆时针旋转

R：右面顺时针旋转 R‘：右面逆时针旋转

U：上面顺时针旋转 U‘：上面逆时针旋转

D：下面顺时针旋转 D’：下面逆时针旋转

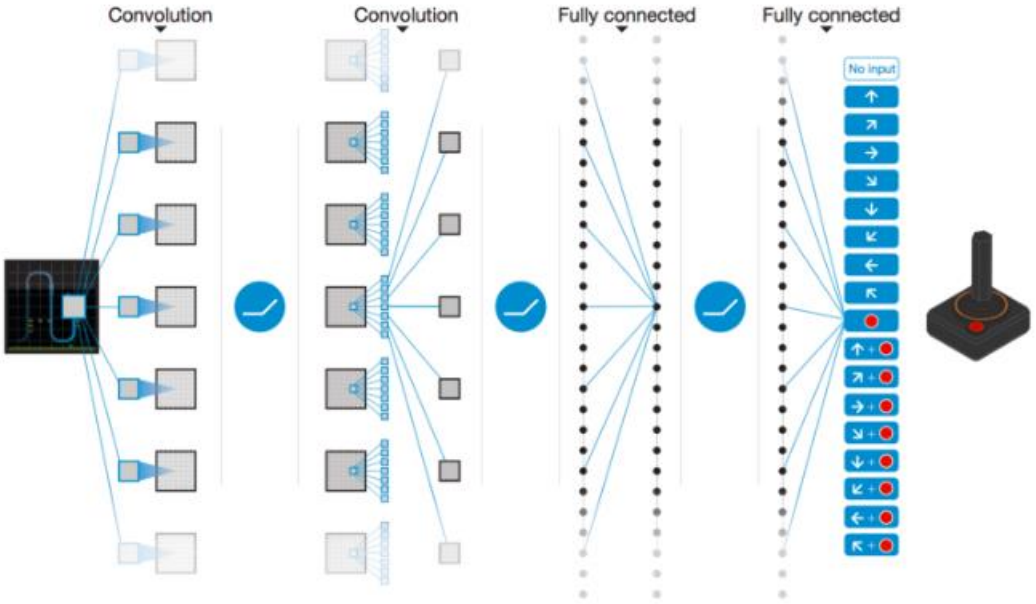
B：后面顺时针旋转 B’：后面逆时针旋转

魔方的单步回报：

每走一步，回报为-1。如果到达了一个之前已经到过的状态，则给予惩罚，回报为-2。

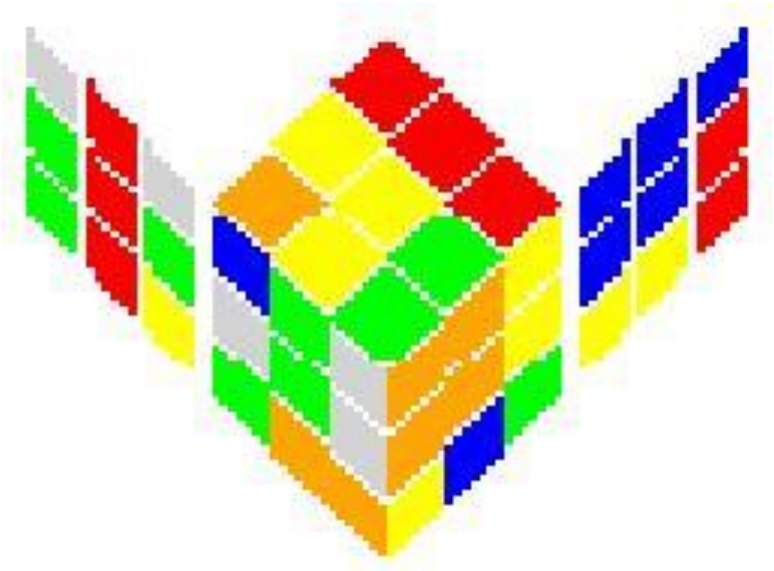
3. DQN（Deep Q-learning Network）模型

对应于 DQN_model.py



经典的 DQN 网络模型如上图所示，网络由 3 层卷积层和 2 层全连接层组成。网络的输入为图像，最终输出的一维向量长度等于动作空间大小。

我们的魔方游戏的动作空间包含 12 个动作，由于我们编写的魔方环境的状态图片与经典雅达利游戏的画面大小不同（渲染图像和训练图像内容相同大小不同），我们修改了卷积层的部分参数，增加了一个全连接层和 ReLU 激活层。



模型关键代码如下：

```
def __init__(self, input_shape, n_actions):
    super(DQN, self).__init__()

    self.conv = nn.Sequential(
        # 3 * 92 * 76
        nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
        nn.ReLU(),
        # 32 * 22 * 18
        nn.Conv2d(32, 64, kernel_size=4, stride=2),
        nn.ReLU(),
        # 64 * 10 * 8
        nn.Conv2d(64, 64, kernel_size=3, stride=1),
        nn.ReLU()
        # 64 * 8 * 6
    )

    conv_out_size = self._get_conv_out(input_shape)
    self.fc = nn.Sequential(
        # 64 * 8 * 6 = 3072
        nn.Linear(conv_out_size, 1024),
        nn.ReLU(),
        # 1024
        nn.Linear(1024, 512),
        nn.ReLU(),
        # 512
        nn.Linear(512, n_actions)
        # n_actions
    )
```

网络输入 3 通道 92x76 大小 RGB 图片，转换为 numpy 浮点数，归一化。

第一层卷积层 Conv1 的 32 个卷积核大小 8x8，步长为 4，输出向量 32x22x18。

第二层卷积层 Conv2 的 64 个卷积核大小 4x4，步长为 2，输出向量 64x10x8。

第三层卷积层 Conv3 的 64 个卷积核大小 3x3，步长为 1，输出向量 64x8x6。

然后将向量展平成长 3072 一维向量经过 3 层全连接层，逐层降维到 1024、512、动作空间长度，最终输出。

第一层卷积层 FC1 为我们增设的层，考虑到输入维度较大。

4. 训练和测试

对应于 Solver.py

程序文档如下：

```
doc = '''
    We start training with the simplest situation (1 step
before solved),
    then train more complex situation (n <- n + 1 until n == 20
steps before solved),
    with model of last step as initial Q_net and target_Q_net.
    Each training step has different hyper-parameters (each
line of TRAIN_PARA).
'''
```

详细如下：

不同于普通的雅达利游戏，魔方在游戏机制上更不易于训练：魔方在还原前，每一步的 reward 都相等（不包括重复到达），而同时，魔方的状态数量及其庞大，如果对一个随机魔方随机初始化，很可能经过极长的时间都无法还原，如果对尝试步数设置阈值，对于大量未成功的尝试，其单步和累计回报都几乎一致，几乎不提供任何状态的真实期望回报的信息，同时也难以通过梯度训练网络。

因此，我们利用了魔方的“上帝之数”定理：任意魔方都可以在 20 步以内还

原。同理，魔方的任意状态都可以从还原状态打乱 20 步达到。为了让魔方可以在合理的时间成本内训练，我们首先训练 DQN 网络还原随机打乱 1 步的魔方，达到一定效果后，再逐步增加打乱次数，最终到打乱 20 步，即完全随机状态。

对于每一次训练，部分超参数有所变化：

```
TRAIN_PARA = [
    # EPOCH_LIMIT = 2e5
    # EPSILON_DECAY_LAST_FRAME = 1e5
    # [rand, max_step, exp_reward, mean_space, epsilon_start,
epsilon_final]
    [1, 10, -1.5, 10, 1.0, 0.1],
    [2, 15, -2, 15, 0.9, 0.1],
    [3, 20, -2.5, 20, 0.9, 0.09],

    [4, 25, -3.5, 25, 0.9, 0.09],
    [5, 30, -4.5, 20, 0.9, 0.08],
    [6, 30, -5.5, 25, 0.8, 0.08],
    [7, 35, -7, 25, 0.8, 0.07],
    [8, 35, -8, 30, 0.8, 0.07],
    [9, 40, -9, 30, 0.8, 0.06],
    [10, 40, -10.5, 35, 0.8, 0.06],
]
```

打乱次数逐层增加时，逐渐放宽训练条件，增加网络可尝试步数，减小期望累计回报。

5. 问题分析

训练中的一个参数对训练结果和效果至关重要：ε-greedy 策略中的 ε。

最初我们令初始 ε 为 1.0，即网络一开始完全随机探索策略，这对于魔方初始状态简单的情况尚可适用，而对于后面几步训练，初始 ε 逐渐降低，网络以一定的概率随机探索，同时也部分参考上步已训练网络的策略建议。

对于单步训练循环，ε 从一开始的较大的值随着魔方操作次数逐渐衰减到 0，即让网络倾向于一开始随机探索魔方的初步策略，后期逐渐转向于参照经过训练的网络执行操作。

虽然我们充分利用了 ε 对网络训练的应用，但整个训练过程中最大的问题也出在 ε 这个参数上。

首先，如上文所说，当 ε 过大时，网络倾向于随机选取动作 action 操作，这时原本需要 n 步还原的魔方只有 1/12 的概率变为（n-1）步还原，而 11/12 的概率变为（n+1）步还原，很显然，当 n 较大时，网络很可能使魔方变换到更加复杂的情况，最终大量训练样本以步数过多终止，其得到的信息却不足以训练网络更好地还原魔方。

当 ε 过小时，网络按照已训练出的模型选择动作，此时对于一部分已训练过的魔方模型可能会得到很好的效果，但相反，对于为探索的训练样本，网络选择的动作和随机选择的动作差别不会特别大，魔方也很可能陷入死循环或者更复杂的状态。这种情况下，极易造成网络模型欠拟合。

因此，我们初步判定，在可接受的训练成本内，经典 DQN 网络可能相对不适合魔方还原游戏，如果要训练魔方游戏，需要对网络结构或者训练过程做大量的改良。

6. 测试

对应于 Play.py 准确率和错例参见 fail.txt 保存模型见 models 文件夹

我们目前训练的最好的模型，对于打乱 10 步的验证集数据，成功率仅 2%-4%，且对打乱更少步数的验证集数据准确率也不如前步的模型。尽管如此，我们的模型对于随机打乱 6 步以内的魔方已经有较好的还原能力。