

1 逃生游戏

1.1 游戏介绍

给定一个随机的5×5的地图，包含一个终点（emerald_block），若干障碍（sea_lantern），以及可以蔓延开的火（fire），目标是让史蒂夫能学会走到终点并尽量不要碰到火。

1.2 DQN

1.2.1 DQN的本质

DQN的本质是将Q-Learning中的Q函数的存储形式由表格转变为神经网络，以适应状态或动作连续、复杂的应用场景。

对于Q函数而言，我们每向其输入一个（状态，动作）对，Q函数就会输出其相应的估值。在Q-Learning中，我们以状态为纵轴，以动作为横轴，构建一表格，以实现这一映射。

但表格法有这样的問題：

- ①只能处理离散分布的输入，不能处理连续分布的输入。当状态或动作是连续变化的值的时候，就会有无数多的状态或无数多的动作，此时表格便是不可行的；
- ②即使输入是离散的，但当可能的状态、可行的动作过于复杂时，表格也会极为庞大，其存储会占据极大空间，也是不实用的。

那么为什么DQN用神经网络代替表格后，就能解决以上问题呢？

- ①神经网络天然得能处理连续的输入值；
- ②神经网络的本质也是一个函数，对其的训练就是调整其参数使之能够拟合我们的目标。请注意，这些参数是为所有输入所共享的，也即我们只用一定的参数量就能实现对所有（状态，动作）对的估值的计算——与之相对的，Q-Learning其实相当于为每一对（状态，动作）对都做了一个独立的到估值的映射，因为对于每一个（状态，动作）对，Q表中都会为其填写一个估值。总而言之，估值计算的参数共享让DQN能够使用更小的空间来表示Q函数。

那DQN有无缺点呢？

当然有。由于神经网络的参数为所有输入的（状态，动作）对所共享，因此在拟合其中某个输入对时，对另一输入对拟合的误差可能会变大——捡了这个、丢了那个。我们是如何权衡的呢？像所有的神经网络一样，去计算一批样本的某种平均误差指标，再进行反向传播更新参数，以平衡对各种不同输入对的关照程度。

DQN的神经网络结构的设计极为重要。设计得太过简单，DQN就无法很好地拟合Q函数；设计得太过复杂，泛化能力就会降低。

DQN对reward的设计提出了高要求。不合理的reward设置会导致神经网络的反向传播中出现梯度爆炸或梯度消失。一种经验性的设计方式是将reward设计在一个较小的、在0附近的区间内，如 $[-1, 1]$ ，以防止出现极大的数，导致梯度爆炸；对于此时梯度消失，可以通过引入残差解决。

本实验中基于tensorflow框架设计的神经网络结构如下：

```

1 conv1 = tf.contrib.layers.conv2d(X, 32, 8, 4, activation_fn=tf.nn.relu)
2 conv2 = tf.contrib.layers.conv2d(conv1, 64, 4, 2, activation_fn=tf.nn.relu)
3 conv3 = tf.contrib.layers.conv2d(conv2, 64, 3, 1, activation_fn=tf.nn.relu)
4 flattened = tf.contrib.layers.flatten(conv3)
5 fc1 = tf.contrib.layers.fully_connected(flattened, 512)
6 self.predictions = tf.contrib.layers.fully_connected(fc1, len(actionSet))

```

十分经典的三个卷积层后接两个全连接层了。其中X为以史蒂夫为中心获取的9×9的地图（经过一定预处理，如将方块的字符名映射到数字）。输出为上、下、左、右各自的概率。

本实验的reward设置为：

行为	奖励
移动一步	-1
死亡	-10
找到终点	200
进入火中	-50
未死亡但超时	-5

1.2.2 两大技巧：replay buffer与fixed target

replay buffer

由于游戏的连续性，我们按时间采集到的样本也是连续的。如前文所言，所有的输入是共享网络参数的，又由于按时间采集到的连续的状态彼此之间十分相似，如果直接把这些连续的样本输入网络训练，就会导致网络偏向于这些大量出现的相似样本，而对其他样本的估值不准。

类似的问题，是神经网络训练中的通病；而针对这一通病的解决办法，最常用的便是随机打乱训练样本的数据。此处的replay buffer其实就完成了这样一件事：将历史上采集到的样本存储在样本池中，在每次训练网络时，从样本池中随机选择batch个样本作为训练集，就可以让用于网络训练的样本在样本空间上的分布更加均匀，解决神经网络偏向特定类别样本的问题。

在本实验中，我们用列表replay_memory来充当样本池。其可容纳500000个样本；当包含的样本数超过50时即开始训练（我们训练的一个批次为32）；当充满后，会丢掉最早的样本，以给新采到的样本腾出空间。

fixed target

训练神经网络，需要有样本特征X和样本标签y，但在本DQN的训练过程中并没有现成的样本标签。

怎么办呢？

在传统的Q-Learning中，我们是按贝尔曼方程去做Q表的迭代，从而让agent的表现越来越好——贝尔曼方程可以带领模型变得更好。

$$V(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \times \max_{a'} V(s', a')$$

于是，我们尝试用贝尔曼方程来生成我们的标签。具体做法是，当收到输入样本特征 (s, a) 时，我们可以通过已有的网络去得到公式中所需的 $V(s', a')$ ，进而可得到贝尔曼方程对 (s, a) 的估值，也即样本标签 $V(s, a)$ 。这样我们就有可用的样本标签了。

那fixed target是做什么的呢？

为了让训练过程更加稳定，我们让神经网络参数更新一定次数后，再更新样本标签，这便是fixed target。不用fixed target，target每一轮的位置都不一样，导致网络参数很难朝一个稳定的方向持续更新；应用fixed target，则在下次更新标签前，target的位置都不变，神经网络的参数就能在这样一定轮次内朝着target稳定地更新。

在本实验中，我们用q_estimator代表我们训练的网络，用target_estimator来代表我们的target。每次更新网络参数时，我们把相同的样本分别输入这两个网络，q_estimator输出的即为模型的预测值，target_estimator输出的值用于贝尔曼方程的计算，即可得到样本标签，进而我们可以进行误差计算、反向传播。q_estimator持续更新，target_estimator每隔100轮更新一次（更新时将q_estimator的参数复制给target_estimator即可）。

1.3 不要回头

在做之前的DQN时我们就有亲身体会：这agent整条路上都还学得挺好，就爱在个别的地方反复横跳。当时我的解决办法是不允许agent走回头路——即走到已经走过的地方。这次，我们采用类似的解决办法，不过也要更结合该实验的实际。

迎面而来的最大的实际竟然是同步问题。实验中返回的地图是以史蒂夫为中心的9×9的地图，因此我们无法直接记录史蒂夫的坐标来确定某个地方他是否走过——他始终在中间。但注意到我们的地图中只有一个终点，也即返回的地图中有且只有一个emerald_block，我们通过记录emerald_block的坐标就可以判断史蒂夫到过什么地方了。然而尝试一下后发现，这个方法并不能准确记录史蒂夫到过的地方。将相关信息打印出后发现，原因是这边代码的运行和那边史蒂夫的行为不同步——也即将移动指令输入Minecraft后，那边史蒂夫还没完成这次移动，这边代码就已经又跑好几轮了。解决办法是在向史蒂夫发送移动指令后面加一句time.sleep(0.1)，给史蒂夫时间去完成这次移动（当然事实上这样或者尝试更大的暂停后，还是偶尔会有不同步的情况发生。此处0.1的使用是一种折衷的结果——不同步的情况已经很少发生了，但又不至于让训练过程变太慢）。

在同步问题解决后，每次游戏开始后，我们会初始化一个空列表my_path用于记录史蒂夫走过的地方。史蒂夫每想要走一步时，我们就可以根据他的当前位置算出他的下一位置在哪里——如果下一位置在my_path里，那就不向史蒂夫传指令；如果下一位置不在my_path里，就向史蒂夫发送指令让他移动，并且把这下一位置加入my_path中。（注意这些位置并不是史蒂夫的位置，而是用emerald_block相对于史蒂夫的位置来表示的）。

有一个问题是，如果史蒂夫走到死胡同了怎么办？解决办法是，当检测到史蒂夫连续10轮没有改变位置后，就清空my_path，让史蒂夫可以走出死胡同。

1.4 训练与测试

1.4.1 训练

总共有160张地图。总共进行1000局游戏的训练。

每次训练时，我们从160张地图中随机抽取一张地图并训练10局，于是事实上只有100张图被用于训练。

当然会有 ϵ -greedy的设置，训练过程中 ϵ 从1逐渐降至0.1。

1.4.2 测试

测试仍然是在这160图上进行的。注意到这160张图中还有60张模型是完全没见过的，因此这对我们模型的泛化能力提出了一定考验。

为了提升模型的泛化能力，我们在测试时设置 $\epsilon=0.2$ ，保留一定的随机性。

为了排除偶然因素的干扰，对于每张图，我们给模型两次尝试机会。

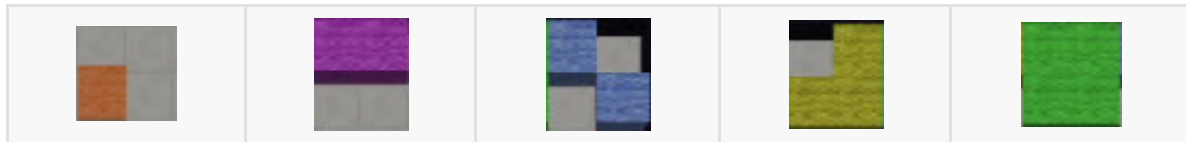
测试过程录制了视频，链接在此：<https://www.bilibili.com/video/BV1ES4y1R7HX>

结果：模型在138张图上取得了成功，高于用于训练的100张图，这说明我们的模型具有一定的泛化能力。

2 俄罗斯方块

2.1 游戏介绍

游戏界面一排五格，共有以下五种方块：



其他与传统俄罗斯方块规则一致。

2.2 Q-Learning

2.2.1 算法理解

Q-Learning是众多采样-估值方法中的一种。

与蒙特卡洛方法进行一轮完成采样再更新Q表相比，Q-Learning每执行一个动作后，都会利用这一动作采集到的样本对Q表进行即时的更新，因此相对于蒙特卡洛方法会更加快捷。

于在策略学习的Sarsa相比，Q-Learning是离策略学习的方法。离策略的好处在于在采样时不使用所训练的策略，而是通过另一个策略来采样，再经重要性采样后送给被训练的策略学习。这样一是消除了被训练的策略与采集的样本间的相关性，从而被采集到的样本可以反复使用；二是可以避免被训练的策略陷入局部最优，因为若直接用被训练的策略来采样，那么当策略学习到一定程度后变得较为稳定，再依此采样得到的样本就会有明显的偏向性，而离策略中用另一个策略来采样就可以缓解这种症状。

总而言之，Q-Learning是off-policy的TD(0)的算法，在强化学习中得到了广泛应用。

Q-Learning的Q表更新公式为：

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

其中 Q 为估值函数（即Q表）， S 为当前状态， A 为所选动作， R 为 S 下执行动作 A 获得的奖励， S' 为 S 下执行动作 A 后到达的状态。

2.2.2 实现

如何在俄罗斯方块中定义状态和动作呢？

一种天然的想法是，当前的游戏池状态和当前即将下落的方块就是状态，而方块旋转几次、在哪个位置下落就是动作。这在理论上的确是可行的。

然而，就游戏池而言，如果将整个游戏池都当作状态存入Q表的话，那状态数目可就太多了！当然，一个可行的方法是利用DQN，这样就能让所有的状态共享参数，从而能够接纳如此多的状态。

但上面逃生游戏用过DQN了，这下还是想尝试下传统的Q-Learning。

回想一下，对于俄罗斯方块这个我们都很熟悉的游戏来说，我们在玩的时候，往往最关心的是最上面几层的状态。基于这种想法，我们不再以整个游戏池为状态，而是只用游戏池目前已堆积方块的层中最上面两层作为状态。获取当前游戏池最上面两层的功能在`get_curr_state`函数中实现。于是一层五格，两层一共十格，每一格的取值只有“有方块”和“无方块”，故所有的状态数只有 $2^{10} = 1024$ ，是以表格形式存储可接受的。






再来看看下落方块和操作。下落方块存在种类的不同，而即使是同种下落方块也存在初始下落方向不同的情况。对于操作而言，存在左移、右移、旋转等。一言以蔽之，描述这些信息是很繁琐的，因此这些信息并不适合直接记录到Q表中。采取的办法是，对于当前的下落方块，我们得到所有可能的操作(s,r)，其中s代表右移s格（如果s为负则代表左移），r代表旋转次数（逆时针）。然后，我们以当前的游戏池搭配每一个可能的操作(s,r)，得到所有可能的下一状态（当然，就记录到Q表而言，我们只关心最上面两层）。这样，我们就将下落方块和操作的信息也转化为了游戏池最顶上两层的信息，进而存储在表格中。这样是有好处的，因为俄罗斯方块中许多不同的方块、不同的操作引起的最终结果是一样的，如果直接记录下落方块和动作，就会让本质相同的状态与动作被记录到两个不同的地方，造成冗余和不准确。取得所有可能操作的函数为get_possible_actions，结合游戏池、当前下落方块和操作推导新状态的函数为pred_insta_drop2。

2.3 启发式算法

2.3.1 半规则式

在应用Q-Learning训练后我们发现，模型效果的提升十分缓慢。因此，我们尝试引入一些人为制定的策略来辅助模型学习。这样的方式被称作启发式。

基于所获得的游戏池最上面两层的状态，我们给出了如下对不同方块的启发式处理：

方 块	启发式处理
	插空即可，且优先插深度最深的空。
	优先看最顶层是否有两个横着的连续的空位，有的话就放在这个位置；否则看有无两个竖着的连续的空位，有的话就放在这个位置；否则交给AI处理。
	查看最顶上两层有无 $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ 或 $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ 的位置（其中1代表有方块，0代表无方块），有的话就将方块调整到适合的方向后放到这个位置；否则交给AI处理。
	查看最顶上两层有无 $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ 或 $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ 的位置（其中1代表有方块，0代表无方块），有的话就将方块调整到适合的方向后放到这个位置；否则交给AI处理。
	优先看最顶层是否有两个横着的连续的空位，有的话就放在这个位置；否则交给AI处理。

注意到，这种启发式规则的本质是在某些情况下直接代替AI做决策，从而缩小的AI所要学习的状态空间，让AI能够专注去学那些人类看来不太好处理的情况。AI的学习状态空间缩小后，模型效果提升速度就能得到加快。

仍然注意到，和模仿学习一样，这种方法的代价减小了AI学习出人类无法想象的优秀策略的可能性。以上的方法只是在我们人类看来最优的或仅仅只是比较好的方法，但我们将其固定死后，就限制了AI在这些状态下发现超越人类策略的可能性。

还需注意到，与下面提到的全规则式不同，这里的启发式算法被称为半规则式。这是因为这里的方法既有人为制定的规则，又有AI学习的参与。实际训练中，该方法的起始平均每局消除层数的值约在10~11之间，但在训练结束后这一值上升到了12，这证明了在这种方法下AI仍是在学习的。

2.3.2 全规则式

一种完全应用人为制定规则的算法，基本思想是宽度为2的方块放在两侧，让宽度为1的方块（单个的方块和两个方块组成的长条）放在中间。

2.4 结果

在训练到趋于稳定后得到数据如下：（平均指所有历史训练数据的平均）

方法	平均每局达到等级	平均每局消除层数
仅Q-Learning	16.5	0.95
半规则式+Q-Learning	41.5	12
全规则式	143	60

录制的视频链接：<https://www.bilibili.com/video/BV1J34y197LY>

视频合集中：

Baseline只用了Q-Learning的方法；

Heuristic使用的是初级的半规则式启发式算法+Q-Learning；

Strong中我们对半规则式启发式算法进行了改进与优化；

MathAngel应用的方法为全规则式。

2.5 反思

我们应该在AI的训练中掺杂多少人为因素？

回想拉开人工智能兴起大幕的阿法狗，其最初的版本是利用人类棋手的棋谱训练并加上强化学习。仅这样的方法，就在与世界冠军李世石的对局中取得了四胜一负的战绩。而后，阿尔法狗的新版本阿尔法零彻底放弃对人类棋谱的学习，全程使用强化学习，变得更加的不可战胜。就阿尔法狗这个例子而言，完全抛弃人类策略的AI更为强大。

现在思考，我们为什么会想到引入人为规则，使用这样的启发式算法呢？

在学习的状态空间过大时，一是不好训练，对模型搭建与训练策略提出更高的要求；二是需要消耗巨大算力。

但启发式算法也有其局限性，也即正如前文所言的，对AI创造出超过人类的策略造成了限制。

在人工智能领域，我们经常遇到许多需要tradeoff的场景——如此看来，在训练AI的过程中引入多少人为因素，也是一个需要tradeoff的事情。

回到本实验，为什么全规则的方法会比其他两种方法好那么多呢？我想有以下的原因：

①这个游戏本身为传统俄罗斯方块的简化版本，从人类角度来看已经是个非常简单的游戏了，因而人类能够设计出表现非常好的策略（例如我们的全规则式）；

②我们的训练时间和算力有限，较少的训练轮数下AI难以达到①中人类所制定策略的水平。

总而言之，是加规则让AI更好训练，还是减规则让AI探索更多可能，我们是需要针对具体场景、具体任务而探讨的。

