强化学习DQN作业实验报告

DQN

实验原理

总览

DQN算法由Mnih[1]等人在2013年提出。DQN与Q学习类似，是一种时序差分算法。并且，DQN和QLearning学习的是最优Q函数，而非SARSA中当前策略的Q函数，这会导致学习稳定性和效率的提高。

> [1] Mnih V , Kavukcuoglu K , Silver D , et al. Playing Atari with Deep Reinforcement Learning[J]. Computer Science, 2013.

DQN中的Q函数

DQN中Q函数的构造与QLearning类似：

$$Q_{\mathrm{DQN}}^{\pi} \approx r+\gamma \max_{a^{'}} Q^{\pi}(s^{'}, a^{'})$$

对于target，有：

$$Q_{tar:\mathrm{DQN}}^{\pi} = r+\gamma \max_{a^{'}} Q^{\pi}(s^{'}, a^{'})$$

在估计$Q_{tar:\mathrm{DQN}}^{\pi}(s,a)$时，DQN并没有使用下一个状态实际采取的动作，而是使用该状态所有潜在可用动作中最大Q值进行估计。

DQN中的动作选择

虽然DQN是离策略的，但DQN中智能体收集经验的策略同样重要。首先是如何快速探索状态-动作空间来增加发现在环境中进行动作的好方法的机会，其次，状态-动作空间的增大，使得DQN需要用神经网络的函数逼近来进行归纳类似地状态与动作。

函数逼近方法的一个重要性质是它对不可见的输入可进行**泛化**。例如，未访问的状态-动作对的Q函数估计值有多好？线性回归、神经网络等函数逼近方法都具有一定的泛化能力，但列表方法却没有这种能力。

考虑$\hat{Q}^{\pi}(s,a)$的表格表示法，假设状态-动作空间非常大，且在训练初期，每个代表特定$\hat{Q}^{\pi}(s,a)$的单元都被设为0，则在训练期间，被访问的状态-动作对被更新，但仍有未被访问的(s,a)有$\hat{Q}^{\pi}(s,a)=0$。由于状态-动作空间很大，许多(s,a)对都不会更新，则这些对应的Q值恒为0。这是因为**表不能了解不同状态-动作之间如何相互关联。**

但是，神经网络可以从已知的(s,a)的Q值推导出未知的$(s^{'},a^{'})$，因为神经网络能推测不同状态-动作是如何相互关联的。这意味着智能体不必访问所有的(s,a)来得到Q函数的良好估值，智能体只需要找一个动作-状态空间的代表性子集即可。

但是，网络的**泛化能力一般是有限的。**一般用VC维衡量网络的泛化能力，而在这里，我们讨论两个与强化学习有关的对网络泛化能力显著影响的因素。首先，如果网络接收到的输入与训练时输入有很大不同，则不太可能产生一个好的输出。一般而言，在训练数据周围的输入空间的较小邻域内，泛化效果更好。同时，强化学习中容易出现**逼近的函数明显不连续，则会导致泛化能力变差**，这是因为神经网络隐式地认为输入空间是局部平衡的。例如悬崖行走，则具有不连续的奖励函数，这就导致神经网络的泛化能力可能变差，导致最后生成的路径可能不会沿着悬崖边行走。

经验回放

如果使用当前策略收集的数据来更新策略参数，则每个经验只使用一次。当与梯度下降法学习的函数逼近方法结合时，会出现问题。首先每个参数的更新必须很小，因为梯度值传递当前参数值周围小区域内关于下降方向的有意义的信息。然而，某些经验的最佳参数更新可能会很大。这种情况下，需要使用经验对网络参数进行多次更新，才可以利用传达的所有信息。

DQN这样的离策略算法使用经验后并不需要丢弃已经使用过的经验，这点最早由Long-Ji Lin[2]发现，并提出了一种对Q学习的增强，称为**经验回放**。他观察到，由于强化学习的试错机制，导致TD学习是缓慢的，并且需要随着时间推移反向传播信息。加快TD学习学习的速度意味着要么加快信用分配的过程，要么缩短试错的过程。经验回放通过促进经验重用来专注于后一种方法。

> [2] Lin L J . Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching[J]. Machine Learning, 1992.

经验回放内存存储智能体近期收集到的k个经验（本实验$k=2000$），如果内存是满的，那么最旧的经验会被丢弃，为新经验腾出空间（即FIFO）。智能体每次训练时，都会从经验回放中随机、均匀抽取一批次或多批次数据，每个批次数据又将一次用于更新Q函数网络的参数。通常，k远大于批次中的元素数量。

内存的大小应该适中。过小会导致不能容纳很多事件的经验，过大则不能让每条经验在丢弃前被多次采样，使得学习效率降低。

使用FIFO，则是因为较旧的经验在学习中变得不再特别有用。在有限的时间和计算资源下，根据局部性原理，更应学习最近收集的经验。

目标网络

本次实验中，利用目标网络计算Q_{tar}^{π}。Mnih[3]使用该方法以稳定训练。在DQN中，Q_{tar}^{π}是不断变化的，因为它取决于$\hat{Q}^{\pi}(s,a)$。训练中，调整Q网络的参数θ是为了使$\hat{Q}^{\pi}(s,a)=\hat{Q}^{\pi_{\theta}}(s,a)$和$Q_{tar}^{\pi}$之间的区别最小，但当$Q_{tar}^{\pi}$变化时，这会变得很困难。

> [3] Volodymyr M , Koray K , David S , et al. Human-level control through deep reinforcement learning[J]. Nature, 2019, 518(7540):529-33页.

为了减少Q_{tar}^{π}在训练步骤间的变化，我们引入目标网络。目标网络是带有参数ϕ的第二个网络，也是$\hat{Q}^{\pi_{\theta}}(s,a)$的滞后副本。目标网络是用来计算$Q_{tar}^{\pi}$的，更新式子如下：

$$Q_{tar:\mathrm{DQN}}^{\pi_\theta} = r + \gamma \max_{a'} Q^{\pi_\theta}(s, a')$$

$$Q_{tar:\mathrm{DQN}}^{\pi_\phi} = r + \gamma \max_{a'} Q^{\pi_\phi}(s', a')$$

ϕ会周期地更新为θ的当前值，被称为替换更新。在本次实验中，ϕ的更新周期是100。

目标网络为什么能稳定训练？首先，每次计算$Q^{\pi_\theta}(s,a)$时，由参数表示的Q函数都略有不同，因此对应相同的状态-动作对，$Q^{\pi_\theta}(s,a)$可能完全不同。这种Q值的偏移，会破坏训练的稳定性，因为它使得网络应该尝试逼近的值变得不清晰。引入目标网络是为了阻止目标的移动，使策略分化或摇摆的可能性减小。

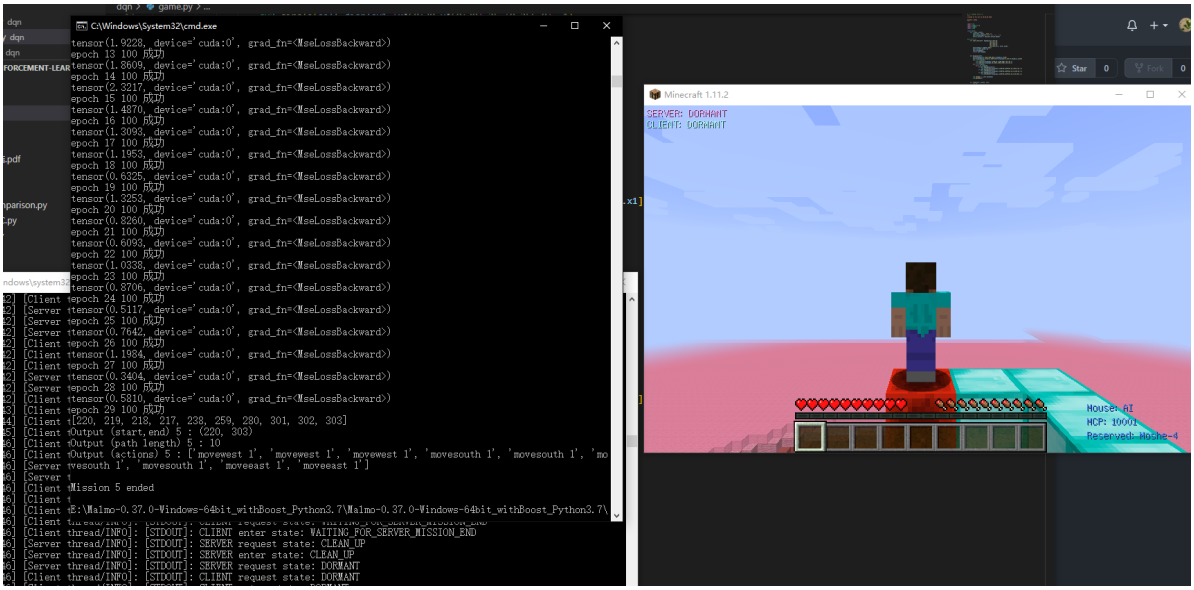
DQN算法

伪代码如下：

```
初始化
for m = 1 to MAX_STEPS do
    利用当前策略并存储2000经验池(s,a,r,new_s)
    for b = 1 to BATCH_SIZE do
        从经验回放内存中抽取批次b的经验
        for u = 1 to BATCH_SIZE do
            for i = 1 to BATCH_SIZE do
                计算Q值
            end
            计算损失
            更新参数
        end
    end
    if m mod REPLACE_ITER == 0 then
        目标网络参数 = 估计网络参数
    end
end
```

结果与分析

程序正常运行，并正确输出。



由于开始时模型不能正确收敛，本人按**xxx**同学的教程在windows上下载了Malmo平台，并成功运行之前实验所用代码，并将课程使用的train.py里使用CPU进行训练的代码改为可使用CUDA加速的代码。但发现依然不能收敛，此处与同学讨论发现，train.py中`q_target = b_r + GAMMA * q_next.max(1)[0]`处应该做reshape，否则会触发PyTorch的广播机制，导致`q_target`与我们的预期不符，最终不能很好收敛。这里感谢**xxx**同学的提醒。

最终利用原代码可以正常运行，通过调参和简化网络，最后得到的网络（见上一部分）在本机绝大多数情况下能在约10s收敛并正确输出结果。本机使用平台为Intel core i7-9700处理器与NVIDIA GeForce RTX 2070(laptop)。

本次实验使用的网络：

第一层卷积层，输入通道为3，输出通道16，核大小5*5，无补0。

第二层为全连接层，输出通道4。

由于实验十分简单，而为了能够更快的收敛，导致本次实验使用网络非常小，并且该网络性能较好。

本次实验还曾尝试使用任意点为起点，以探索更多的状态-动作对。但这样就需要遇到空气就死，否则会在“孤岛”上死循环。但是这样的设计，导致最后的网络收敛到智能体摆烂一心求死，因为这样惩罚更少。我也曾尝试改变每个奖励的数值，则会导致智能体在必死的“孤岛”上来回蹒跚，降低了收敛速度，因此放弃，每次仍选择固定起点，同时这样也可以让网络接收到的输入与训练时输入一致，能有更好的泛化能力。

本次实验也曾尝试切换损失函数，因为PyTorch官网在训练DQN时使用HuberLoss，但在更改后发现，HuberLoss使得最终的收敛速度有所下降，因此作罢。

最终在几次尝试后，train文件并无太多变化，只是改变了epsilon贪心策略为软策略，并且修改了Qtarget网络的替换次数，以及修改了最后的网络结构。