强化学习入坑指南

强化学习的入门

1.强化学习的圣经《Reinforcement Learning : An Introduction》

2.由上海交通大学*ACM*班创始人余勇教授团队编写的《动手学强化学习》，及配套网址

3.B站课程西湖大学的强化学习课程

4.*xqy*学习路线

- 从*Q-learning*入门，学习*Q-learning*的数学证明，*Q-learning*的数学本质为泛函分析中的压缩映射，当映射次数达到一定数量时，能够映射为*S-A*空间的不动点，即为Q^*,也有*Q-learning*的证明论文，请点这里
- 学习完*Q-learning*后，学习*On-policy*的基础算法-*PG*算法，*PG*算法的公式推导：知乎*up*苍溪的文章。并在这里搞清楚$Q_\pi(s,a),V_\pi(s)，\pi(a\mid s)，P(s\prime\mid s,a)，\eta，P_\pi(s_t)，Pr(s_0\rightarrow s_t，\pi)$的含义，这些在强化学习学习过程中非常重要，强化学习经常被建模为$M=(S,A,T,\eta,R)$，分别为：状态空间、动作空间、状态转移函数、折扣因子、奖励函数。
- 学习完*on-policy*和*off-policy*的经典算法之后，可以开始看*DQN*的实现方法，可以参考这篇论文。*DQN*相较于*Q-learning*,最主要的差别在于其状态的输入是连续的，而使用神经网络来拟合Q函数是一个明智的选择，但同时也会带来Q难以收敛的问题，文中使用两个Q网络进行更新，首先固定一个*Target_Q*,使用*Eval_Q*进行探索，探索得到足够的样本后进行更新，只更新*Eval_Q*，当经历了几次更新后，将*Eval_Q*的神经网络参数值传递给*Target_Q*,完成两个网络的更新。
- 这个时候已经学习了*Q-learning*、*PG*、*DQN*等算法，已经能够熟练处理一些状态离散或连续，但动作离散的问题场景。这个时候可以学习一下*Actor-Critic*网络，*PG*算法中还是使用的蒙特卡洛方法进行预期折扣奖励的估计，这种估计期望准确而方差较大，而*DQN*中没有使用策略网络进行一个动作的探索，而是使用的一个探索因子。因此，*Actor-Critic*应允而生，一个学习过程下来，仍然可以学习到最优的Q^*,并且可以证明，此时得到收敛的$\pi^*(a|s)$收敛到$\max_a Q^*(s,a)$。
- 这个时候已经学习了*on-policy*的*Actor-Critic*网络，而*Actor-Critic*的真正用途在*off-policy*中，因此接下来学习这篇论文,这篇论文给出了*off-policy*的数学近似证明。
- 接下来可以学习强化学习中的最强算法：*PPO*，学习*PPO*可以直接从知乎*up*苍溪的文章看证明，然而*up*的文章有很多细节没有披露，这篇关于*TRPO*的文章给出了更加详细的数学证明。是的，*PPO*是从*TRPO*引申出来的。而在代码复现时，*PPO*使用了一种叫做*GAE*(广义优势估计)的方法来估计优势函数，*GAE*的数学推导看这篇文章。
- 当学习完*PPO*算法后，可以开始关注一下*SAC*算法，*SAC*算法与*off-policy*的*Actor-Critic*算法只在奖励上有差距，记*off-policy*的*Actor-Critic*中的奖励函数为$R(s,a,s')$,则记*SAC*中的奖励函数为$\overline{R}(s,a,s')=R(s,a,s')-\alpha log(\pi(a|s))$,因为这个改变，会改变$\overline{Q}^*$,由原本$Q^*(s,a)=\sum_{s'}P(s'|s,a)(\sum_r P(r|s,a,s')r+\eta\sum_{a'}\pi(a'|s')Q^*(s',a'))$变更为$\overline{Q}_1^*(s,a)=$

$\sum_{s'} P(s'|s,a)(\sum_r P(r|s,a,s')(r-\alpha log(\pi(a|s))+\eta\sum_{a'}\pi(a'|s')\overline{Q}_1^*(s',a'))$。同样地，*soft-Q*还有另外一种定义，$\overline{Q}_2^*(s,a)=\sum_{s'} P(s'|s,a)(\sum_r P(r|s,a,s')r+\eta\sum_{a'}\pi(a'|s')(\overline{Q}_2^*(s',a')-\alpha log(\pi(a'|s'))))$,可以证明：$\overline{Q}_1^*(s,a)=\overline{Q}_2^*(s,a)-\alpha log(\pi(a|s))$,而在实际代码中，经常使用的是$\overline{Q}_2^*(s,a)-\alpha log(\pi(a|s))$这种形式。这里不需要看论文，直接代码实操。并且在连续动作中还使用到了一种叫做重参数化技巧的方法，能够大大降低梯度方差，这后续再谈，重点理解一下*SAC*为什么不用重采样来保持分布的一致性。

- 当对*SAC*也了解的差不多的时候，可以看一下连续动作的*DQN*的延申-*DDPG*，*DDPG*可以看这篇文章,而关于*DDPG*的证明看这个网址,*DDPG*的精髓在于学习一个$a=u(s)$,这是一种动作的选择策略，可以证明，对于这样的一种策略，可以找到一个收敛的$Q^u(s,a)$。此外，在代码中，*DDPG*将随机性加在了动作上，即为输出的动作添加一个期望为0，方差逐渐减小的正太分布随机变量，这样保证了探索。
- 在学习了连续动作的*DDPG*算法以后，可以对*off-policy*的策略梯度算法如：*PPO*、*SAC*如何输出连续动作进行一定的了解。既然是连续动作，那动作就会服从一个连续的概率分布，这个概率分布通常没有先验知识，因此假设为正太分布、*Beta*分布、*Gamma*分布等都可以，策略网络输出的是这些分布的参数，在选择动作时，使用这些参数确定的随机分布进行采样。这部分直接看代码即可。
- 对于多智能体强化学习的入门，可以看这篇文章，这篇文章讲了一些基础的多智能体强化学习的符号表示和博弈类别。
- 对于多智能体非合作有限博弈，会存在一个纳什均衡，纳什均衡的证明可以看这篇文章，这篇文章即使数学没有很高的水平也可以看懂。
- 对于代码方面的学习，推荐使用知乎up苍溪的github库。
- 事实上，我们应该区分马尔可夫过程有齐次和非齐次之分，马尔可夫过程本身所指的是:$P(s_t|s_{t-1}s_{t-2}...s_0)=P(s_t|s_{t-1})$,齐次马尔可夫性才有进一步的条件:$P(s_t|s_{t-1})=P(s_1|s_0),\forall t\geq 1$。即状态转移时不变性。

不可约非周期齐次马尔可夫过程的稳态

记有*n*个状态的马尔可夫随机过程的状态转移矩阵为*T*，*T*的具体形式如下：

$$T=\begin{vmatrix} p(s_1|s_1) & p(s_1|s_2) & ... & p(s_1|s_n) \\ p(s_2|s_1) & p(s_2|s_2) & ... & p(s_2|s_n) \\ ... & ... & ... & ... \\ p(s_n|s_1) & p(s_n|s_2) & ... & p(s_n|s_n) \end{vmatrix}$$

可以证明，*1*一定是*T*的一个特征值，因为$[1,1,1,...1]$是T^T的特征向量，特征值为*1*。记*T*的特征向量为$[x_1,x_2,x_3,...,x_n]$,有$Tx_i=\lambda_i x_i$,将x_i进行归一化使得$||x_i||_1=1$，则$||Tx_i||_1=||\lambda_i x_i||_1=|\lambda_i|||x_i||_1=|\lambda_i|$。由范数不等式：$||Ax||_1\leq||A||_1||x||_1$,其中$||T||_1=1$,则可以证明$|\lambda_i|\leq 1$。因此转移矩阵*T*的特征值最大为1，最小为-1。对于任意的初始状态概率分布向量$p_0=c_1x_1+c_2x_2+...c_nx_n$,$\lim_{n\to\infty} T^np_0=c_ix_i+(-1)^nc_jx_j$,即达到一个周期平稳状态，如果*T*没有特征值为-1,则达到一个平稳状态，即$\lim_{n\to\infty} T^np_0=c_ix_i$其中系数$c_i$并不重要。

重参数化技巧(Reparameterization Trick)

在算法连续动作的SAC算法中，使用到了一种叫做重参数化的trick，这种方法可以使得从策略网络输出的参数生成的分布中采样可以带参数的梯度，数学原理为：$X \sim f(x), Y = g_\theta(X) \sim h_\theta(y)$，可以得到：$\int_y h_\theta(y)m(y)dy = \int_x f(x)m(g_\theta(x))dx$,因此$\bigtriangledown_\theta \int_y h_\theta(y)m(y)dy = \int_y h_\theta(y) \bigtriangledown_\theta log(h_\theta(y))m(y) = \int_x f(x) \bigtriangledown_\theta m(g_\theta(x))dx$,做实验验证，这种方法求得的梯度相较于Reinforce的方法具有更小的方差。

离散动作确定性策略和随机性策略中的action-mask机制

在强化学习中，在智能体选择动作时人为制定一些简单规则进而避免绝对劣势的动作输出，能够加速智能体学习，避免现实智能体做出一些很反常理的动作。Python中经常使用到masked_fill()函数

- 确定性策略: 以DQN为例，在选择动作时，有两种方案。一种是随机选择，另一种是根据状态-动作函数选择最大化价值的动作，因此，我们在选择动作时，将不能选择的动作使用mask进行屏蔽，代码如下:

```
if np.random.rand(1) < self.epsilon:
  random_action = np.random.rand(self.act_shape)
  random_action = torch.tensor(random_action, dtype=torch.float32)
  random_action = random_action.masked_fill(~legal_act, 0)
  act = random_action.argmax(dim=1)
else:
  logits = model(feature)
  logits = logits.masked_fill(~legal_act, float(torch.min(logits)))
  act = logits.argmax(dim=1)
```

  这样我们在动作选择方面便进行了mask，而我们对这些动作进行mask即表示这些动作是绝对的劣势动作，我们不希望智能体采取这些动作，我们也不希望智能体更新这些动作的价值，即这些动作在计算状态-动作价值函数时也应该被排除在外。因此，我们在更新Q函数时也应该将mask考虑在内，代码如下:

```
  with torch.no_grad():
      q= model(next_batch_feature)
      q = q.masked_fill(~next_batch_obs_legal, float(torch.min(q)))
      q_max = q.max(dim=1).values
  target_q = rew + self._gamma * q_max * not_done
  logits= model(batch_feature)
  loss = torch.square(target_q - logits.gather(1, batch_action).view(-1)).mean()
```

- 随机性策略: 随机性策略由于引入了策略函数，因此在mask上与确定性策略有所不同。首先，在动作选择上，由于随机性策略使用策略网络输出动作的概率分布，因此mask需要对动作的概率分布进行屏蔽，代码如下:

```
with torch.no_grad():
  probs = model(state)
  probs = probs.masked_fill(mask,torch.finfo(torch.float32).min)
  probs = F.softmax(probs - torch.max(probs,dim = 1),dim = 1)
  action = torch.multinomial(probs,1).item()
```

而在训练时的屏蔽机制对于不同的算法有着不同的规定。PPO算法在更新价值函数时直接计算$V(s)$而不是$Q(s,a)$,因此这里不用引入mask。而在更新策略网络时，由于需要计算$log(prob)$,因此在这里需要重新引入mask机制，代码如下:

```
    prob = self.actor.pi(s[index], softmax_dim=1)
    prob = prob.masked_fill(s_mask[index],torch.finfo(torch.float32).min)
    prob = F.softmax(probs - torch.max(probs,dim = 1),dim = 1)
    entropy = Categorical(prob).entropy().sum(0, keepdim=True)
    prob_a = prob.gather(1, a[index])
    ratio = torch.exp(torch.log(prob_a) - torch.log(old_prob_a[index]))
```

而SAC算法在训练时有更多的地方需要引入mask机制，价值网络的更新代码如下:

```
    with torch.no_grad():
        next_probs = self.actor(s_next)
        next_probs = next_probs.masked_fill(next_mask,torch.finfo(torch.float32).min)
        next_probs = F.softmax(next_probs - torch.max(next_probs,dim = 1),dim = 1)
        next_log_probs = torch.log(next_probs+1e-8)
        next_q1_all, next_q2_all = self.q_critic_target(s_next)
        min_next_q_all = torch.min(next_q1_all, next_q2_all)
        v_next = torch.sum(next_probs * (min_next_q_all - self.alpha *next_log_probs),
dim=1, keepdim=True)
        target_Q = r + (~dw) * self.gamma * v_next
```

策略网络的更新代码如下:

```
    probs = self.actor(s)
    probs = probs.masked_fill(next_mask,torch.finfo(torch.float32).min)
    probs = F.softmax(probs - torch.max(probs,dim = 1),dim = 1)
    log_probs = torch.log(probs + 1e-8)
    with torch.no_grad():
        q1_all, q2_all = self.q_critic(s)
    min_q_all = torch.min(q1_all, q2_all)
    a_loss = torch.sum(probs * (self.alpha*log_probs - min_q_all), dim=1, keepdim=True)
    self.actor_optimizer.zero_grad()
    a_loss.mean().backward()
    self.actor_optimizer.step()
```

折扣系数奖励的普遍性

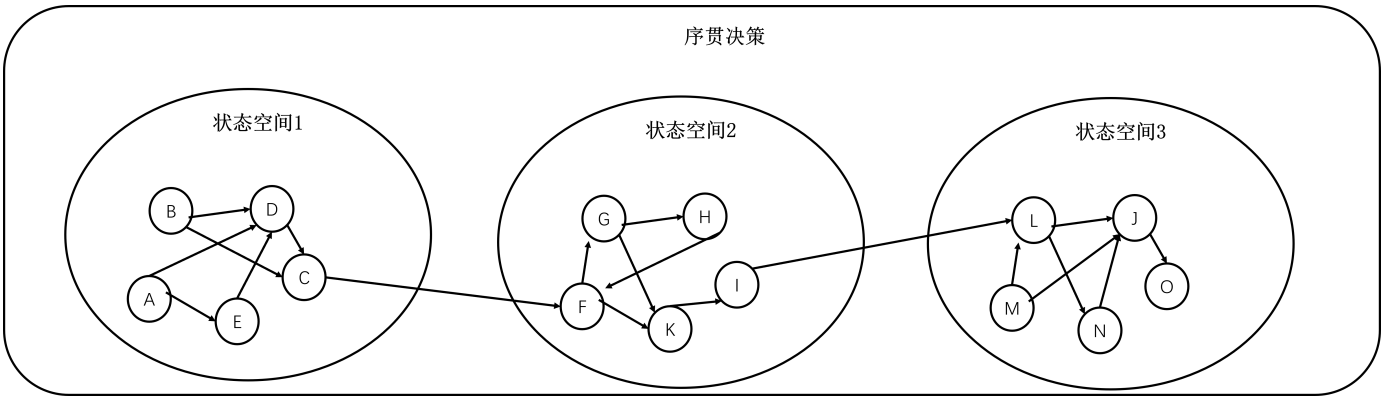
在经济学领域，同样存在折扣系数。举个例子，假如现在有人要给你钱，一种方法是现在当场给你100元，第二种方法是过一年以后给你一百元。毫无疑问，第一种相比第二种价值更高，因为未来是充满风险的，假设到未来他给钱的概率是P，则拿到的钱的期望为$100P < 100$。因此人在做出决策时，会很自然的把未来收益通过一个折扣项计算到当下的收益中。但这个折扣系数是否如强化学习中所讲，是一个固定的值，那也未必，下面这篇论文 从另一个方面讲述了人类在实际决策时更偏向的折扣奖励系数，并为强化学习如何模仿人类学习提供指导。

竞赛经验(reward shaping,HER)

1. 关于在不改变最优策略的情况下，如何在reward的设置上进行人类知识注入的数学理论，详情见吴恩达的论文。这篇论文给出了如何设计人为添加奖励来帮助智能体快速找到最优策略。总的来讲，这篇论文给出的指导是在设置人为添加奖励时，不应与智能体的动作有关，而只与当前状态和下一次状态有关，并且在函数形式上设置成势函数的形式，即用下一次状态的势函数（乘以一个折扣项，折扣项与强化学习中的设定一致）减去当前状态的势函数，这样就能保证最优策略的不变性。
2. 阶段奖励，即在当前episode所处的不同阶段给予不同的奖励，在腾讯智能体初赛中很受用，经过分析，这与初赛的状态-动作空间的结构有很大的关系。

- 首先，智能体初赛是一个寻找宝箱的强化学习场景，如这张图：

智能体需要从左下角的菱形方块处移动到右上角的终点处，期间如果能拿到宝箱则获得宝箱奖励，而最大奖励往往需要拿到全部宝箱。智能体的观测能够观测到自己距离各个宝箱的距离，如果某个宝箱被取走，则这个距离会被判定为999。经过分析，这是一个序贯决策，在拿到第i个宝箱之前，智能体的状态在第i个状态空间中来回切换，而当取到第i个宝箱的那一刻，智能体的状态从第i个状态空间跳转到第i+1个状态空间。这样整个任务可以被分割成若干个只有一个动作进行连接的子任务，那就可以根据子任务设置子阶段奖励。状态-动作空间表示如下面这张图



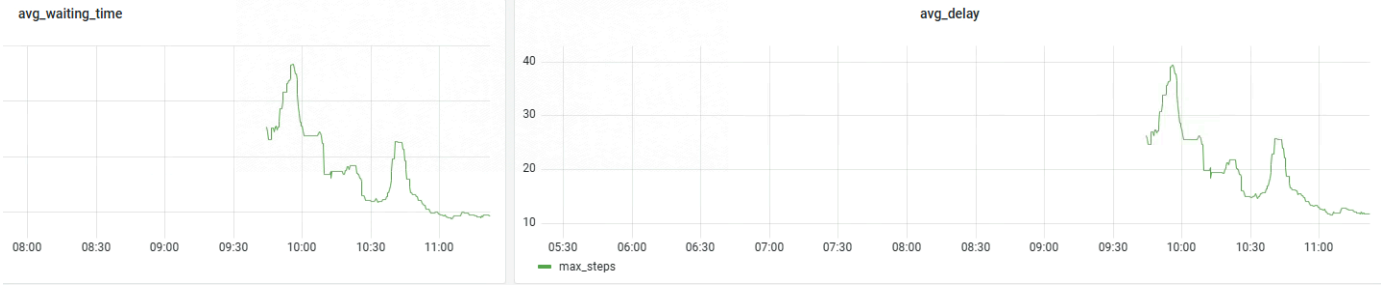
这张图中，在每个状态空间中，黑色箭头表示由于动作而发生的状态转移，在每个状态空间中，只有一个状态采取某个动作才会转移到下一个状态空间。这种结构的强化学习问题适用于阶段奖励。

- 强化学习任务可以分为有终止状态和无终止状态，有终止状态可以看作是有吸收态的无终止状态。有终止状态正如其名，当智能体进入到这个状态后触发terminate，在状态更新上，在终止状态上，价值函数的更新不再添加终止态状态的价值的折扣项，终止态采取任何动作还是进入终止态，终止态无奖励，因此其状态价值函数为0。
  有终止态的强化学习任务奖励可以进行如下设置：由于其在进入终止态后可以等价为获得的奖励时时刻刻都为0，则从初始态开始采取动作时，每次都给一个-1的奖励，则在理想的情况下，智能体会学到到终止态的一个最短路径，即最快速地到达终止态。但往往这种奖励设置是稀疏的，在智能体第一次随机尝试到终点之前，它会迷失在负奖励之中。因此，在此基础上，当智能体靠近终点时，给一个+1的奖励，这样智能体便有更大的可能性能够到达终点，进而学习到终点的奖励。这种奖励的设置原理如下：由于在终止态中获得奖励一直为0，假设折扣系数为1，则如果想要智能体以最快的速度前往终点，则在中间过程得到的奖励不可以大于0，在初赛中，在不同的阶段，这等价于将宝箱设置成终点。

3. 交通信号灯😀

- 交通信号灯的路口图如图:

性能指标有三个：①：车辆延误；②：车辆排队长度；③：车辆等待时间。但在训练过程中，发现车辆延误和车辆等待时间具有强相关性，如下图所示：



而车辆排队长度和车辆等待时间的相关性并没有那么强，此处不再放图。

- 可以通过神经网络拟合出总得分的三个权重系数，分别得到三个权重系数（权重系数9，权重系数10，权重系数8）为：$4.8686, 12.4086, 11.9068$。由此看出，可以重点关注车辆等待时间，并以此作为奖励。

- 下面介绍三个tricks：
  - 车辆等待时间简单建模：假设当前车辆的总的等待时间为:$W_{total} = W^{new} + W^{old}$,其中$W_{total}$为当前所有车辆的等待时间。$W^{new}$为当前新来的车辆的等待时间。$W^{old}$为当前时刻已经在等待的车辆的等待时间。当信号灯选择执行一个动作时，假设将南北直行的灯变绿，那东西方向和南北左转的车辆就会进入一个等待，此时不仅仅会有在信号灯决策的当时时刻已经在进口路口的车，还有在这段时间内陆陆续续进来的车，这种类型的车称为新来的车辆，在下一个决策时刻，新来的车辆的等待时间不作为奖励的参考。一般来讲，使用差分奖励往往能得到一个较好的效果，但如果将差分奖励设置为:$R^t = W^t_{total} - W^{t+1}_{total}$,效果并没有想象中的那么好，因为新来的车辆的等待时间会影响智能体的判断，智能体没有办法判断哪些状态转移是由他的动作直接导致的。因此，差分奖励设置成$R^t = W^t_{total} - W^{t+1}_{old}$，实验验证了这种方法的有效性。

- $state-abstraction$: 通过将每条车道的车的等待时间相加到一起，再按照红绿灯的通行车道合并车道中车的等待时间，再根据new和old的划分，可以将状态空间压缩至24维，大大减少输入维度，并通过实验验证这种方法的有效性。
- 通过分析，如果信号灯每次都产生一个很长时间的绿灯，这样在计算总得分时，并不占优，因此给交通信号灯的绿灯时间设置固定时长，而对于排队较长的车道，应使交通信号灯学会是否切换相位，而每次切换相位都有5s的黄灯时长，尝试使用固定5s的绿灯时长，实验同样验证了这种方法的有效性。

4. HER（Hindsight Experience Replay)是一种目标条件强化学习的算法，非常适用于机器人领域。考虑如下的一个场景，要移动机械臂到桌子上某个固定的点，在设置奖励时，只有当机械臂进入这个固定的点附近某个小领域内才给+1的奖励，这是一个稀疏奖励场景。通常来讲，很难训练，但换个思路，机械臂在移动过程中会经过很多点，这些点在本质上没有区别，只是由于考虑的场景，他们不是我们在预期设置的目标点。但如果在训练时，将这些点的位置加入到策略网络的状态输入中作为目标，并给这些点添加预期奖励+1，就好像机械臂移动过程中遇到的点也是预期点，然后通过**神经网络的泛化性**，模型就能学会怎么找到最开始的预期点，原文论文在这里Hindsight Experience Replay 。
5. 现就一个复杂网格环境的降维经验进行一些探讨，现在假设有一个12*12的网格环境，每个网格中有随机数字(0-20)，智能体扮演一个背着背包的玩家，这个玩家在每个网格处有五种动作：向上走、向下走、向左走、向右走、拾取数字，玩家背包没有限额，有21个格子，每个数字会进到对应的格子中，当格子中某个数字的个数等于四个，会自动消掉，消掉以后玩家会得到一个奖励。当玩家把地图上所有的数字全部消掉以后会获得一个一百的奖励。此外，玩家每走一步会有一个小惩罚，惩罚大小与背包中的数字的个数有关。玩家最多可以走576步。
   - 初步分析这个问题，在每一步都有5个动作，一共可以走576步，那便有5^{576}个可能状态，如果让智能体纯探索，智能体甚至探索不到一个最次优的路径，即每走一步都拿走一个数字，这样既简单又省心，但次优路径毕竟是次优路径，如何降低问题的复杂度，让强化学习能够应用进来?
   - 这个问题可以进行如下的建模，假如我是一个智能体，我的目标是把144个数字全部拿走，那我整个任务可以分成144个子任务，我先拿哪个数字，再拿哪个数字，已经拿过的数字不需要再次拿取，使用硬性mask掉。然后数字之间的移动可以完全交给规则去做。这样问题的复杂程度就变成了$144!$，通过这种方法，可以使智能体学到东西。

State Abstraction

1.状态抽象可以用来简化状态空间的复杂性，提升收敛速度，使智能体能够高效地学习，但状态抽象会把一部分状态映射成相同的状态。倘若原始状态具有马尔可夫性，映射后的状态是否仍然存在马尔可夫性呢，下面这篇论文 给出了状态抽象仍保证马尔可夫性的充分必要条件。
2.这里有一篇论文 汇总了当下各种各样的状态抽象的类型。
3.关于深度双模拟抽象 和Inverse Model抽象的区分: 简单以转移函数为例，抽象后的状态转移函数为$P(z_{t+1}|z_t,a_t),z_t=\phi(s_t)$如果确保抽象后的状态也满足马尔可夫性，就要证明:
$P(z_{t+1}|z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)=P(z_{t+1}|z_t,a_t)$,或者条件更强一点，要满足齐次马尔可夫性，则满足:$P(z_{t+1}|z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)=P(z_{t+1}|z_t,a_t)=P(z'|z,a),\forall t\geq 0$。现在分别从双模拟抽象和Inverse Model抽象两个角度来验证这些条件。

- 双模拟抽象: $P(z_{t+1}|z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)=\frac{\sum_{s_{t+1}\in z_{t+1}}\sum_{s_t\in z_t}P(s_{t+1},s_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)}{P(z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)}=\frac{\sum_{s_{t+1}\in z_{t+1}}\sum_{s_t\in z_t}P(s_{t+1}|s_t,a_t)\pi(a_t|s_t)P(s_t,z_{t-1},a_{t-1}...z_0,a_0))}{\pi(a_t|z_t)P(z_t,z_{t-1},a_{t-1}...z_0,a_0)}=\sum_{s_t\in z_t}P(z_{t+1}|s_t,a_t)\frac{P(s_t,z_{t-1},a_{t-1}...z_0,a_0)}{P(z_t,z_{t-1},a_{t-1}...z_0,a_0)}$,双模拟的条件有些严格，即$P(z_{t+1}|s_t^1,a_t)=P(z_{t+1}|s_t^2,a_t),\forall s_t^1,s_t^2\in z_t$。则$P(z_{t+1}|z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)=P(z_{t+1}|s_t,a_t)=P(z_{t+1}|z_t,a_t),\forall t\geq 0$,直接满足最基本的齐次马尔可夫条件。因此双模拟抽象也被称为最严格的抽象，但理论却是最优雅的。但这种抽象具有一定的不确定性，倘若有一类抽象函数将所有状态映射成一种状态，双模拟抽象状态仍然是满足的。
- Inverse Model抽象：双模拟抽象通过严格的抽象条件保证了抽象后的状态转移服从齐次马尔可夫性，而Inverse Model抽象则从公式的后半段入手: $P(z_{t+1}|z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)=\sum_{s_t\in z_t}P(z_{t+1}|s_t,a_t)\frac{P(s_t,z_{t-1},a_{t-1}...z_0,a_0)}{P(z_t,z_{t-1},a_{t-1}...z_0,a_0)}$,记$B_t^t=\frac{P(s_t,z_{t-1},a_{t-1}...z_0,a_0)}{P(z_t,z_{t-1},a_{t-1}...z_0,a_0)}$,如果$B_t^t=B_t^0=\frac{P(s_t)}{P(z_t)}$,那抽象后的状态满足马尔可夫性，但注意，不一定满足齐次性。记$T_{t+1}^{t+1}=P(z_{t+1}|z_t,a_t,z_{t-1},a_{t-1}...z_0,a_0)$。对$B_t^t$进行展开,$B_t^t=\frac{P(s_t,z_{t-1},a_{t-1}...z_0,a_0)}{P(z_t,z_{t-1},a_{t-1}...z_0,a_0)}=\frac{\sum_{s_{t-1}\in z_{t-1}}P(s_t,s_{t-1},a_{t-1}...z_0,a_0)}{T_t^tP(z_{t-1},a_{t-1}...z_0,a_0)}=\frac{\sum_{s_{t-1}\in z_{t-1}}P(s_t|s_{t-1},a_{t-1})P(s_{t-1},a_{t-1}...z_0,a_0)}{T_t^tP(z_{t-1},a_{t-1}...z_0,a_0)}=\sum_{s_{t-1}\in z_{t-1}}P(s_t|s_{t-1},a_{t-1})\frac{P(s_{t-1},a_{t-1}...z_0,a_0)}{T_t^tP(z_{t-1},a_{t-1}...z_0,a_0)}=\sum_{s_{t-1}\in z_{t-1}}P(s_t|s_{t-1},a_{t-1})\frac{B_{t-1}^{t-1}}{T_t^t}$,下面有意思的是:如果$B_t^t=B_t^{t-1}$可以推出$T_{t+1}^{t+1}=T_{t+1}^t$,而$B_{t+1}^{t+1}=\sum_{s_t\in z_t}P(s_{t+1}|s_t,a_t)\frac{B_t^t}{T_{t+1}^{t+1}}=\sum_{s_t\in z_t}P(s_{t+1}|s_t,a_t)\frac{B_t^{t-1}}{T_{t+1}^t}=B_{t+1}^t$。更直观地讲: $B_t^k=B_t^{k-1}\Longrightarrow B_{t+1}^{k+1}=B_{t+1}^k$。使用一个矩阵表示这个式子:

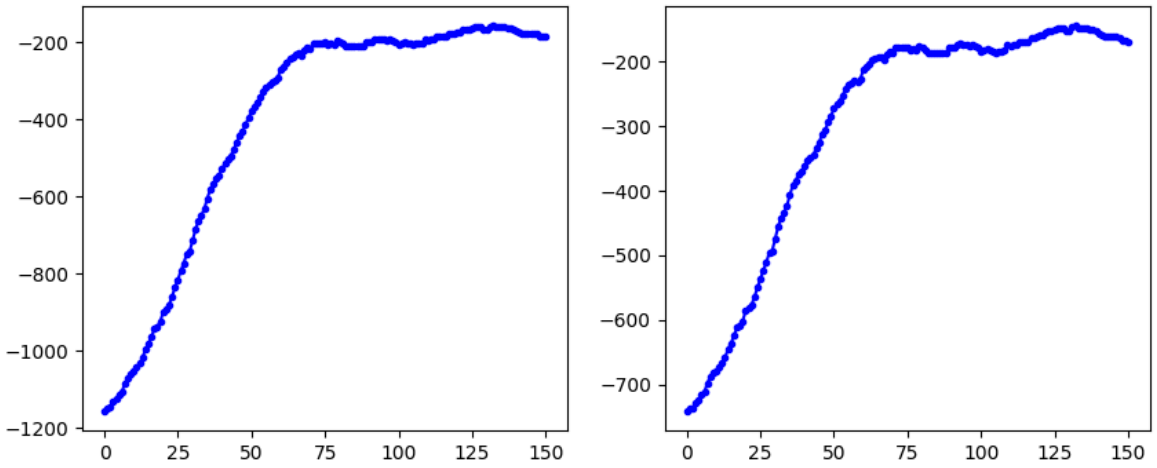
$$\begin{bmatrix}B_0^0 & & & & & & & & & \\ B_1^0 & B_1^1 & & & & & & & & \\ B_2^0 & B_2^1 & B_2^2 & & & & & & & \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & \\ B_t^0 & B_t^1 & B_t^2 & ... & ... & ... & ... & ... & ... & B_t^t\end{bmatrix}$$

可以证明，即当$B_t^0=B_t^1,\forall t\geq 1$时，$B_t^0=B_t^t,\forall t\geq 0$。这便是Inverse Model的推导式，但我们无法证明这种状态抽象是否满足马尔可夫齐次性，值得一提的是，Inverse Model的抽象条件推不出双模拟抽象，双模拟抽象条件也无法推出Inverse Model条件。这两者是截然不同的两种抽象路径。但双模拟抽象却可以保证最优解的不变性。

4.使用双模拟抽象进行的一个小实验：

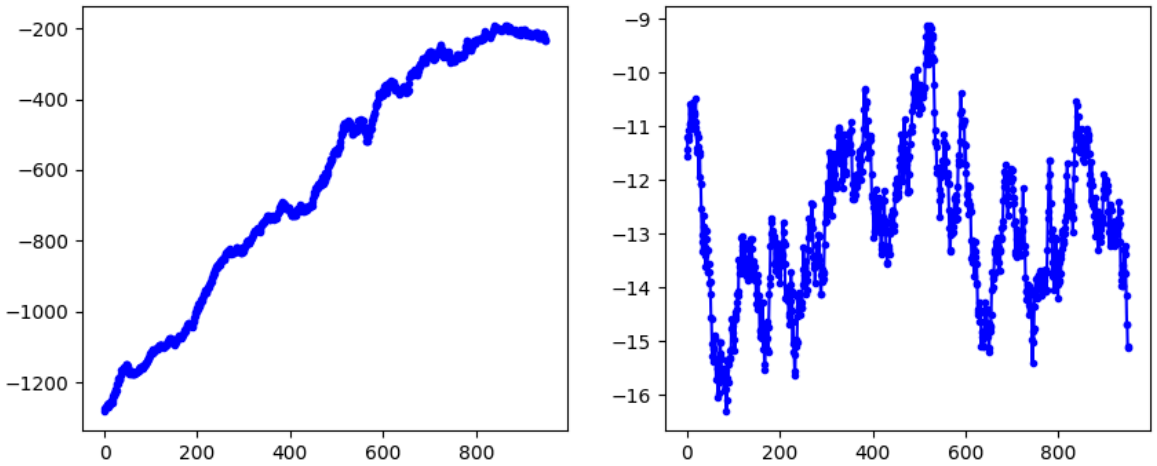
- 竖直杆环境给了一个三维的状态空间，动作空间为一维。其状态数值大都在-1到1之间。给倒立摆环境状态加上干扰，通过扩充状态空间，在原有的状态向量之后添加0到1之间的随机值并扩充到28维。并利用状态抽象将该28维的状态抽象到3维。同时增加噪声的强度重复实验，下面是进行的实验结果。

- 使用SAC算法对没有添加噪声的环境进行学习，学习效果如图:



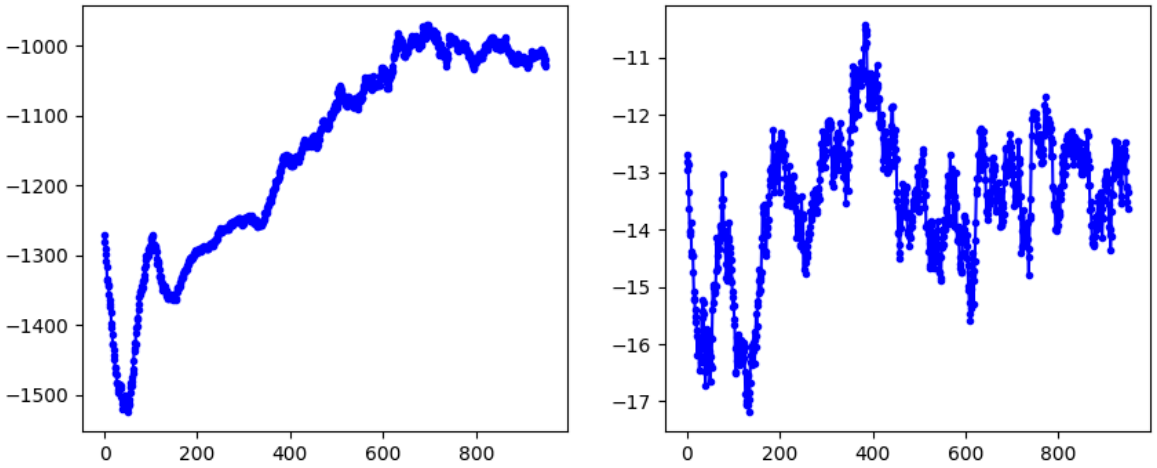
左图为累计奖励，右图为累计折扣奖励，可以看出模型在50轮左右开始学到东西，在100轮左右模型收敛。

- 使用SAC算法对添加小噪声的环境进行学习，学习效果如图:



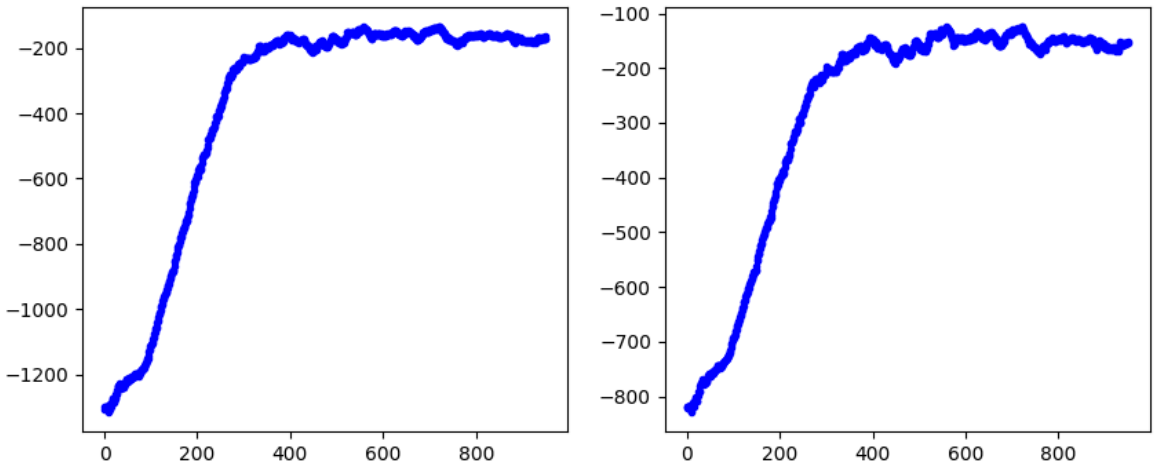
其中左图为累计奖励，右图为累计折扣奖励。其相较于无噪声的环境模型，其学习能力大大减弱，以至于在1000轮左右才出现了收敛的迹象。

- 使用SAC算法对添加大噪声的环境进行学习，学习效果如图:

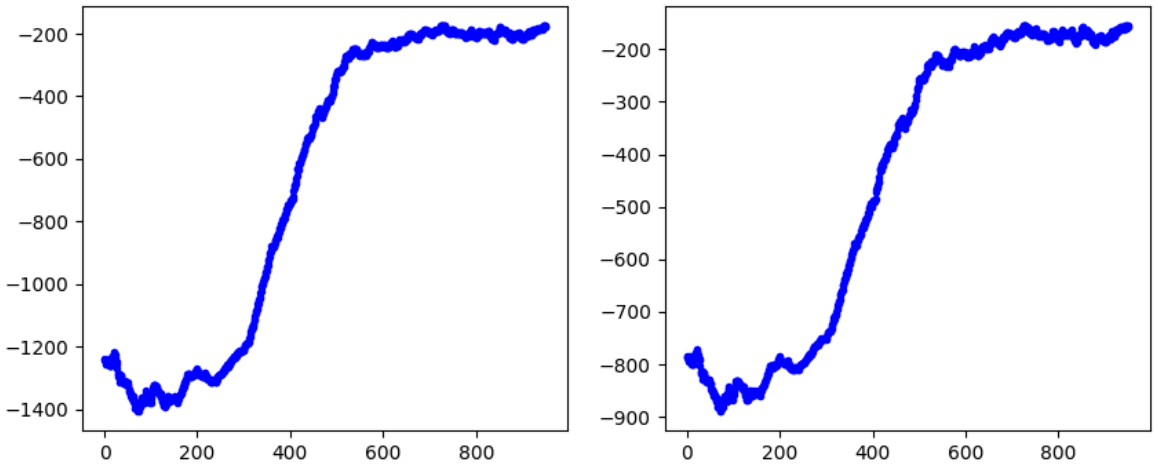


由图可以看出，模型在1000轮几乎没有学到东西，但能看到一个在学习的小趋势。

- 然而使用了双模拟状态抽象以后，情况会有大不同。下面对小噪声环境进行状态抽象，训练结果如图:



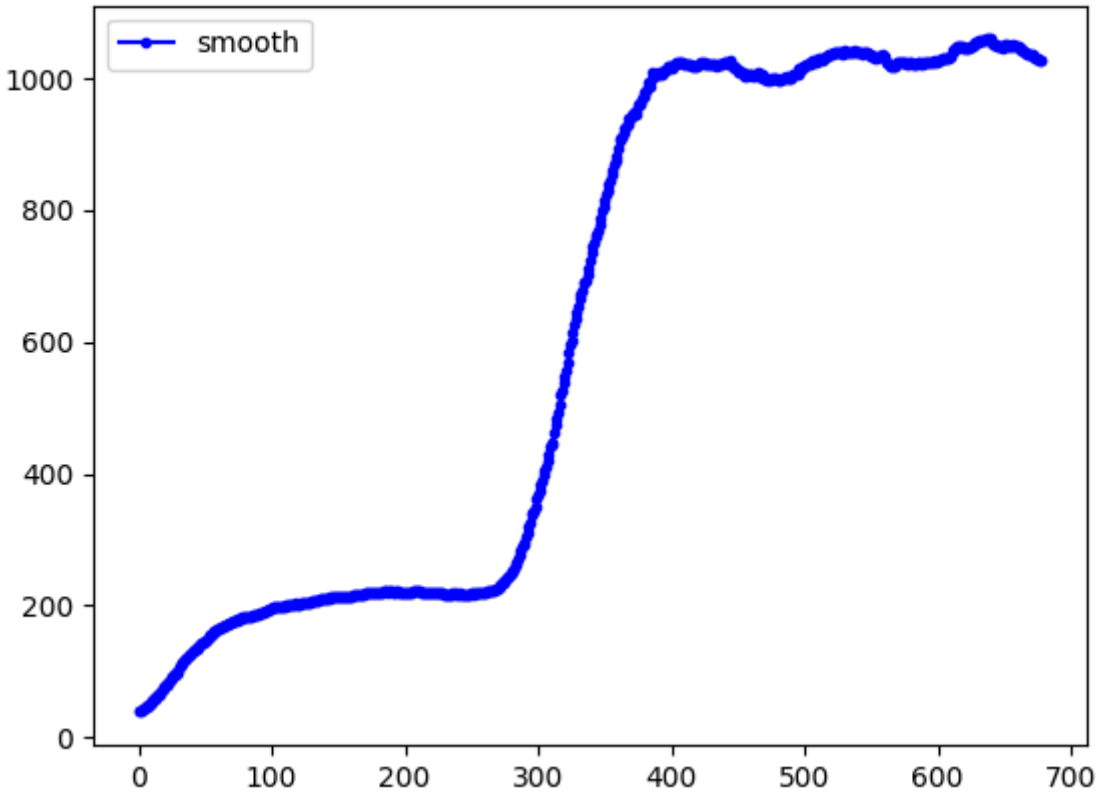
  可以看出，噪声的确让学习变得更加困难，但抽象确实能够很好地抑制某些噪声，剔除了一些无关的状态。换句话说，模型学会了关注他能够改变的状态。
- 同样使用大噪声环境进行实验，实验结果如下图:



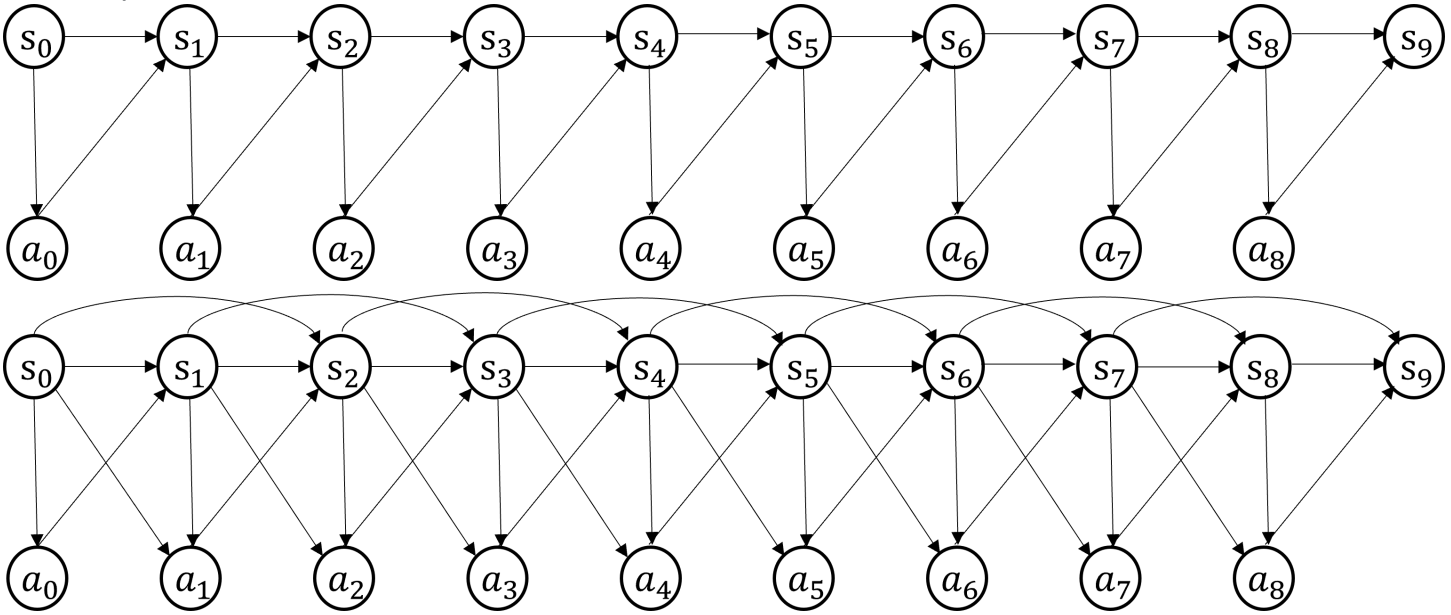
  相较于小噪声环境，大噪声对学习同样产生了影响，但使用了状态抽象后的模型仍然能够在1000轮以内学到东西，并且在学到东西后很快能够收敛到最优解。

POMDP

部分观测马尔可夫过程中观测状态不一定具备马尔可夫性，因此使用常规算法并不能有效地训练模型。以Hopper环境为例，如果将状态只取后六位，此时常规算法很容易陷入局部收敛，即控制小人站着一动不动，这样状态转移和动作就会在一个局部空间中服从马尔可夫分布。训练曲线如图:

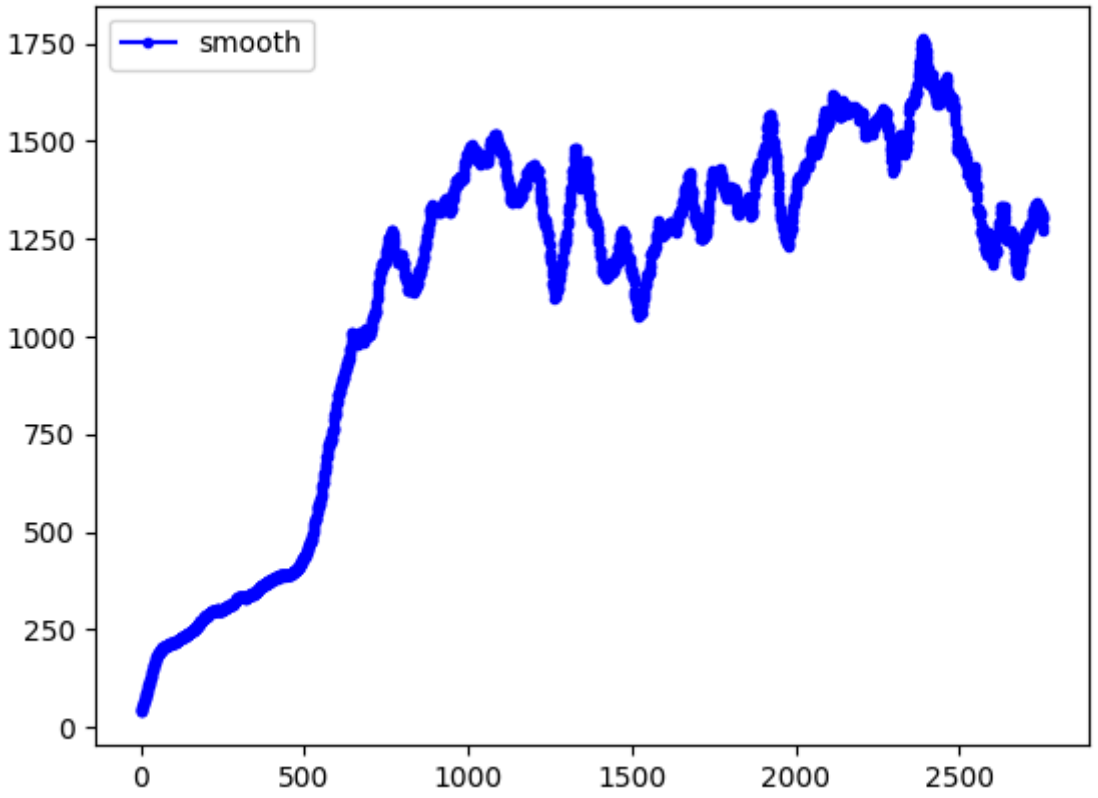


但添加了lstm模块后的Actor和Critic网络能够缓解这一问题,参考这篇论文，即将过去多个时间步的time-step的状态合并视作一个广义状态。下面是一个标准马尔可夫过程和扩展马尔可夫过程的示例:

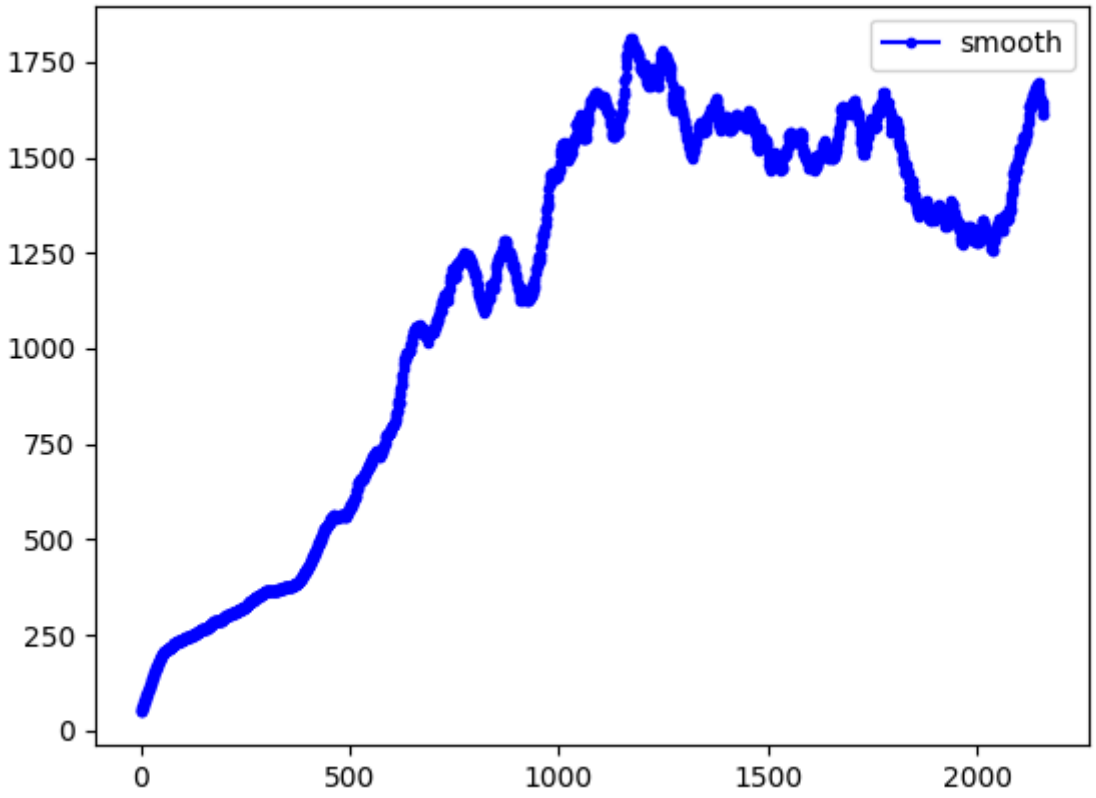


如下图所示的扩展马尔可夫过程，将状态空间和动作空间进行扩展，用如下的方程对扩展马尔可夫过程进行表示: $s_{t+1} = f(s_t, s_{t-1}, a_t), a_t = g(s_t, s_{t-1})$即$x_t = (s_{t-1}, s_t), b_t = a_t$,可以得到$x_{t+1} = f(x_t, b_t), b_t = g(x_t)$,扩展后的状态和动作服从马尔可夫过程。

采用这种方法在Actor和Critic中加入lstm模块，并合理控制观测窗口(即取多少个time_step作为联合状态)。下面是观测窗口为10时的训练曲线:



可以看出，模型能够突破局部收敛，但问题是模型很难收敛，因为观测窗口只是我们假设的，真正的服从马尔可夫过程的观测窗口是不确定的，因此窗口设置为10也不一定满足马尔可夫过程，但可以看出，其相较于常规算法，还是有很大的进步。
下面是观测窗口为32时的训练曲线:



当输入维度增加时，训练效果稍微滞后于窗口为10的模型学习，学习到后面仍然振荡而不收敛，总体性能略强于窗口为10。并且采用lstm模块具有一定的抗干扰性能，当输入观测窗口为35和30时，仍然能够输出一个比较好的动作策略。

使用神经网络估计概率密度函数

通常来讲，使用神经网络估计一个样本集中连续随机变量的概率密度函数是个很困难的事情，因为没有办法保证输出的概率密度在随机变量取值空间积分为1。并且传统机器学习使用核函数估计的方法已经能够对概率密度函数进行一个很好的拟合。下面给出一种利用神经网络进行拟合的方法。

- 神经网络搭配softmax层能够很好的解决分类的问题，假设有两个样本集:$\{X_i^p\} \sim p(X), \{X_j^q\} \sim q(X)$,构建一个泛函:$L = -\int p(x)log(m(x))dx - \int q(x)log(1 - m(x))dx$，通过变分法可以证明当$m(x) = \frac{p(x)}{p(x)+q(x)}$时，该泛函能够取到极小值。而$m(x)$和$1 - m(x)$恰好就是softmax层输出的两个维度。因此，构造目标函数为:$L = -\sum_{x_i^p} log(m(x_i^p))dx - \sum_{x_i^q} log(1 - m(x_i^q))dx$,通过这种方法可以训练出一个样本服从$p(X)$分布的概率。
- 对于任意一个样本x_i，神经网络输出的比值为:$\frac{m(x)}{1-m(x)} = \frac{p(x)}{q(x)}$。如果已知$q(x)$的具体函数形式，可以通过相乘求出$p(x)$的值。
- 通过上述思路，给定一个样本集$\{X_i\}$是服从一个未知分布的样本集，手动构造一个有特定分布的样本集$\{Y_i\}$，如均匀分布。通过上述方法训练两个样本分布之间的比值，然后在求特定值的概率密度时，通过乘以均匀分布的概率密度求得。