- DQN算法实现注意事项

DQN算法实现注意事项

DQN和Q-Learning算法实现有以下相同点:

1. 都是一边与环境交互一边学习的过程, 只能从真实采样的轨迹中学, 而不是事先给定样本或者环境转移函数信息
2. 智能体在学习时的策略都是ϵ-greedy策略以促进探索, 而评估时都可以令$\epsilon = 0$以充分利用策略

不同点:

1. DQN使用了replay buffer来存储轨迹, 每次学习时从buffer中随机采样batch_size个样本进行学习. 这样做的好处是利用了随机梯度下降的优势, 使得神经网络的训练更加稳定. 普通Q-Learning可看做是batch_size=1的特殊情形.
2. DQN除了智能体做决策的Q网络(记为Q_θ)外, 还额外使用了目标网络(记为Q_{tar}). 目标网络不参与决策, 只是为了稳定Q网络更新.
3. DQN的loss公式为

$$L(\theta) = 0.5 \cdot \frac{1}{|B|} \sum_{t \in B} (Q_\theta(s_t, a_t) - y_t)^2$$

其中B表示batch, $y_t = r_t + \gamma \max_{a'} Q_{tar}(s_{t+1}, a')$ 是由目标网络计算出的目标Q值, 而不是由Q_θ本身计算出的Q值.

代码实现提示:

1. 计算目标Q值时, 如果当前时刻t为episode的最后时刻(即 `done=True`), 则改为$y_t = r_t$. 如果不这么做, 可能训练没有效果.
2. 对于CartPole环境, 神经网络不需要过于复杂, 只需2-3层线性+relu激活函数即可取得较好的训练效果.
3. buffer类似于有限容量的队列, 即遵循样本先进先出原则, 可用 `numpy`数组或者 `torch.Tensor`来实现. buffer的随机采样可使用 `np.random.choice`函数实现.
4. loss的计算可使用pytorch中 `torch.nn.MSELoss()`来实现. 计算loss时对于目标网络的输出需使用 `detach()`或者 `torch.no_grad()`, 使得loss梯度回传和更新时只更新Q_θ而不更新Q_{tar}.