强化学习: 作业一

张含笑 502022370062

2022 年 10 月 23 日

1 作业内容

在“蒙特祖马的复仇”环境中实现 Dagger 算法.

2 实现过程

2.1 定义超参数

`args.num_frame`	一共需要采集的游戏画面的帧数, default=2500
`args.num_step`	每次更新训练集采集的步数, default=250
`args.num_epoch`	模型针对更新后输入的训练集 `data_set` 训练的轮数, default = 50
`args.batch_size`	训练模型时参数每更新一次所用的样本数目, default = 50
`args.lr`	模型的学习率, default = 0.05
`args.lr_rampdown_epochs`	学习率降为 0 所需轮数, default = 50
`args.feature_dim`	模型所提取特征的维度, default = 16
`args.beta`	初始专家接管率, 在训练中会随着训练过程逐渐降低, default = 0.5
`args.log_interval`	每次评估模型的间隔轮数, default = 1

2.2 采样与更新数据集

在 `main.py` 中, 定义对 expert actions 的采样函数 `get_expert_action()`, 用于向 expert 展示当前图像, 并采集 expert actions. Expert 的输入可以是 action space 中的 8 个动作, 也可以是 “exit” 或 “bad”.

定义 `beta` 为专家接管率, 以及 `beta_decay()` 函数, 用于在训练过程中逐渐降低专家接管率. 采样时若所获随机数小于 `beta`, 将通过 `get_expert_action()` 函数采样 expert action. 当所采动作为 “bad” 时, 说明游戏失败, 随便设置一个动作即可, 之后不会被加入训练集, 其余的合法 step 会被存入 `valid_step`; 否则通过 `agent.select_action()` 采样.

定义 `num_updates = int(args.num_frames // args.num_steps)`, 收集 `num_updates` 次训练集并对模型进行更新, 每次采样 `args.num_steps` 步. 利用上述方法与环境交互, 每次采样获得的 observation 保存为训练数据, action (除 “bad” 轮次) 保存为对应的标签.

进行 `args.num_steps` 步采样后根据 `valid_step` 将没有被标记为 “bad” 的 step 和对应的 label 存入 `label.txt`.

2.3 定义 MyDaggerAgent

2.3.1 处理训练样本

定义方法 `_build_input()` 用于处理 `dataset['data']`, 若输入图像维度为 3, 则增加一维, 并把 channel 所在的维度置于第二维, 这样处理是为了方便将训练数据以 batch 的形式送入模型, 并对应 `nn.Conv2d` 所需数据维度. 最终训练数据的维度为 `(batch_size, num_channel, height, width) = (args.batch_size, 3, 210, 160)`.

去掉没用的动作, 采样获得的 action 集合为 {1, 2, 3, 4, 5, 6, 11, 12}, 将其映射到集合 {1, 2, 3, 4, 5, 6, 7, 8}, 作为训练模型的 label. 定义字典 `action_map` 和 `action_map_inv`, 用于表示训练模型所用的 label 与 action space 间的双向对应关系.

定义 `_process_label()` 方法用于处理 `dataset['label']`, 将原始训练集中的 action 根据 `action_map_inv` 映射到 label.

2.3.2 策略模型 PolicyModel

策略模型由 `cnn` 和 `action_layer` 两部分构成, `cnn` 用于提取样本特征 (特征维度由 `args.feature_dim` 定义), `action_layer` 是全连接层, 用于对特征分类, 获得在 action space 上的 logits. 策略模型 `PolicyModel` 结构如下:

```
self.cnn = nn.Sequential(
   nn.Conv2d(input_channel_dim, 16, kernel_size=8, stride=4, padding=0),
   nn.ReLU(),
   nn.Conv2d(16, 32, kernel_size=4, stride=2, padding=0),
   nn.ReLU(),
   nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0),
   nn.ReLU(),
   nn.Flatten())

self.action_layer = nn.Sequential(
   nn.Linear(flatten_dim, feature_dim),
   nn.ReLU(),
   nn.Linear(feature_dim, action_dim))
```

2.3.3 更新模型参数

将模型输出 logits 和 label 的交叉熵作为损失函数, `torch.optim.Adam` 作为优化器.

采用 `torch.optim.lr_scheduler.CosineAnnealingLR()` 调整参数的学习率, 学习率经过 `args.num_epoch` 降为 0, 有助于模型的收敛.

将训练数据和标签打乱顺序, 再按照 `args.batch_size` 分成 `minibatch` 送入策略模型, 计算 loss 并更新参数, 用 `running_loss` 记录每轮训练的总损失, 若优于历史最优损失, 则将此时的模型保存至 `best.pth`.

2.3.4 保存模型

定义 `save_model` 方法, 保存 `running_loss` 最小的模型参数, 且每经过 `args.log_interval` 次训练会保存一次模型参数.

3 复现方式

在主文件夹下运行 `python main.py`.

4 实验效果

见图1. 累计奖励在训练早期很快上升, 但随后上下波动. 这种学习曲线不像传统强化学习算法的缓慢收敛, 更符合 Dagger 算法类似于监督学习的本质. 访问专家次数是增长逐渐放缓的曲线, 这是因为训练时需将专家介入率 `beta` 逐渐减小, 而总访问专家次数是介入率与样本量之积的积分.

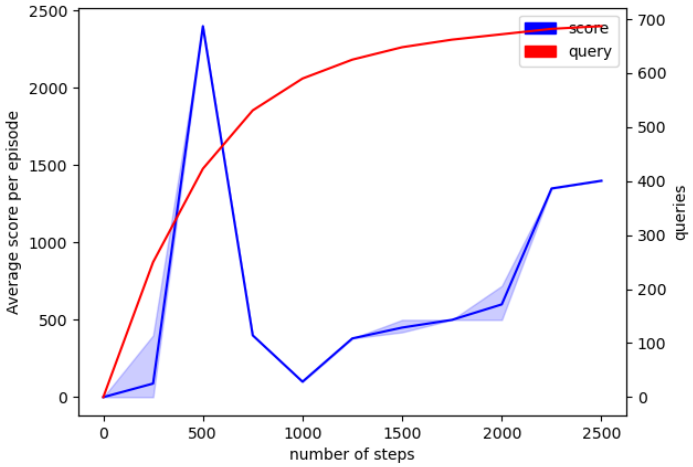

图 1: Dagger 算法

5 小结

在这次实验中, 我发现 Dagger 算法的数据采集过程与传统行为克隆算法极为不同. 由于采集时 behavior policy 为混合策略, 专家不能对智能体有完全控制. 这使得专家策略能在接近于由智能体策略导出的状态分布下被访问, 降低了行为克隆算法中 distributional shift 的问题.

同时这样的混合策略也会带来问题: 如果智能体的策略不够优秀, 混合策略的探索能力会比纯专家策略更差. 如果在某个关键状态由习得的策略而不是专家给出动作, 智能体很可能错失具有高回报的动作. 对此的一种改进方法是: 引入一种 “接管函数”, 专家仅在关键状态接管并给出指导. 如果控制好专家接管率, 其 distributional shift 和 Dagger 算法一样可以得到控制, 同时具有更好的探索性能.