强化学习：作业一

史浩男 502024370026

一、作业内容

在“蒙特祖马的复仇”环境中实现Dagger算法。

> 算法关键思想：不仅学习专家在给定状态下的动作，还要探索由自己策略带来的状态，增加对这些状态的学习机会。

二、实现过程

首先修改Dagger.py文件，实现了MyAgent

算法核心使用 `resnet18` 神经网络架构，输出类别为8个动作。 `transform` 负责图像的预处理，将输入状态转化为适合模型的224*224形式

设置每次训练10个epoch，学习率为0.02

```
class MyAgent(DaggerAgent):
    AdrianNeuer *
    def __init__(self):
        super(DaggerAgent, self).__init__()
        self.batch_size = args.num_steps
        self.transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
        ])
        self.epochs = 10
        self.num_classes = 8
        self.learning_rate = 0.02

        self.model = resnet18(num_classes=self.num_classes).to(device)
        self.optimizer = optim.SGD(
            self.model.parameters(), lr=self.learning_rate, momentum=0.9)
        self.loss_fn = nn.CrossEntropyLoss()
```

然后我分别实现了update和select_action方法，详见代码

对于主体代码，我做了部分优化：

- 首先是将默认设置中的 `MontezumaRevengeNoFrameskip-v0` 替换为了更新版本的 `MontezumaRevengeNoFrameskip-v4`

```
def get_args():
    parser = argparse.ArgumentParser(description='RL')
    parser.add_argument(
        *name_or_flags: '--env-name',
        type=str,
        default='MontezumaRevengeNoFrameskip-v4')
```

- 在手动试玩部分的代码，接受输入不够鲁棒，这里我增加了一个函数`get_action`用于重写，使得无论怎样的非法输入，都不会使程序退出

```python
def get_action(action_shape):
    # 检查输入的动作是否是数字，如果不是则提示重新输入
    # 使用一个循环来确保输入是合法的动作
    while True:
        action = input('Please input action: '.format(action_shape - 1))
        try:
            # 尝试将输入转换为整数
            action = int(action)
            # 检查动作是否在有效范围内
            if 0 <= action < action_shape:
                break  # 输入合法，退出循环
            else:
                print("Action out of range! Please input a number between 0 and {}.".format(action_shape - 1))
        except ValueError:
            # 捕获转换失败的情况（输入非整数）
            print("Invalid input! Please enter a valid integer.")
    return action
```

- 在main.py的类Env中，增加了索引“【0】”防止代码报错

```python
def reset(self):
    return self.env.reset()[0]
```

- 为了防止Dagger算法带来的数据集规模越来越大，占用资源和开销越来越多的问题，我设置了数据集最大规模为10000，达到后会随机丢弃100的数据量

```python
for i in range(num_updates):
    print("NUM_UPDATES: {}.".format(i + 1))
    if len(data_set['data']) == 10000:
        indice = np.random.choice(len(data_set['label']), size: 1000, replace=False)
        data_set['data'] = np.delete(data_set['data'], indice, axis=0)

        data = data_set['data']
        data_set['data'] = []
        for arr in data:
            data_set['data'].append(arr)

        data_set['label'] = np.delete(data_set['label'], indice, axis=0).tolist()
```

专家策略模块

`expert`部分定义了专家策略模型`PolicyModel`，用于为Dagger算法提供参考策略。

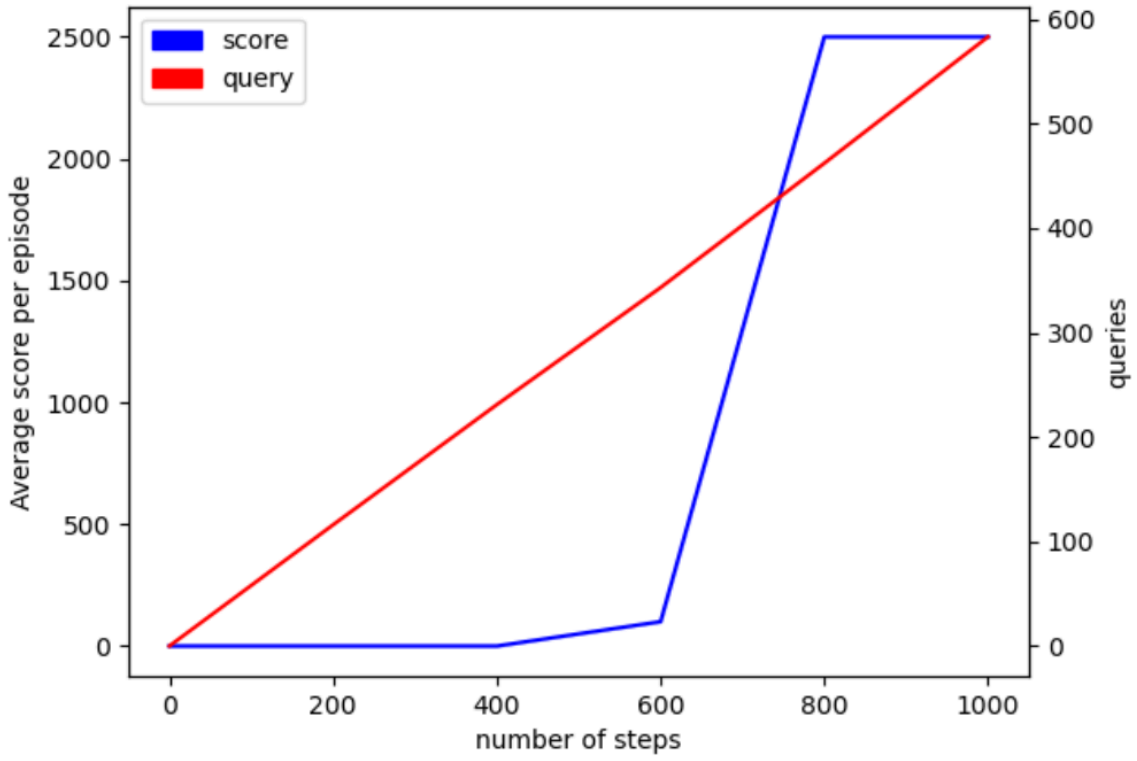
```python
class PolicyModel(nn.Module, ABC):
    new *
    def __init__(self, state_shape, n_actions):
        super(PolicyModel, self).__init__()
        c, w, h = state_shape
        conv1_out_w = conv_shape(w,  kernel_size: 8,  stride: 4)
        conv1_out_h = conv_shape(h,  kernel_size: 8,  stride: 4)
        conv2_out_w = conv_shape(conv1_out_w,  kernel_size: 4,  stride: 2)
        conv2_out_h = conv_shape(conv1_out_h,  kernel_size: 4,  stride: 2)
        conv3_out_w = conv_shape(conv2_out_w,  kernel_size: 3,  stride: 1)
        conv3_out_h = conv_shape(conv2_out_h,  kernel_size: 3,  stride: 1)
        flatten_size = conv3_out_w * conv3_out_h * 64
        # 定义卷积层和全连接层
        self.conv1 = nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1)
        self.fc1 = nn.Linear(in_features=flatten_size, out_features=256)
        self.fc2 = nn.Linear(in_features=256, out_features=448)
        self.policy = nn.Linear(in_features=448, out_features=self.n_actions)
```

该模型使用卷积神经网络来处理输入状态，对Env进行了重新包装，得到特征后通过全连接层计算动作概率和值函数，最终输出的动作概率用于采样最优动作，从而指导智能体的行为。

三、复现方式

主文件夹下运行 python main.py.

四、实验效果



五、思考题

在玩游戏的过程中标注数据与Dagger算法中的标注数据方式有何不同？这个不同会带来哪些影响？

在传统的数据标注和训练过程中，游戏的数据标注是先进行的，即玩家完成游戏后生成并标注所有数据，再使用这些标注数据进行训练，标注和训练是完全独立的，训练数据也不会在训练过程中新增，因而满足独立同分布，各个数据的权重相等且不变。

Dagger算法采取了不同的策略，训练的过程中动态标注数据，每轮采样时部分数据通过专家的指导被标注，然后加入训练样本集，使得训练数据集随着探索和训练不断扩充。这种方式使得训练数据的生成具有序列性，前期的探索和学习会影响后续的数据标注和生成，从而导致训练数据的分布和权重发生变化，不再是静态且独立同分布的。数据是先通过模型探索产生再进行标注，标注行为本身不会影响状态的生成。虽然这种方式使得算法的效果收敛较慢，但因其探索了更多的状态，减少了过拟合的风险。

六、小节

本实验的实现方式和真正意义上的Dagger还有一定区别，主要体现在以下几个方面：

1. 初始化：本实验中，我们使用专家模型来对初始状态集合进行标注，从而构建初始训练集，并不是利用专家的状态-动作对初始化策略π。
2. 交互：基本符合，记录自己策略和专家策略的动作，形成专家标注
3. 数据集拓展：基本一致，将智能体执行策略所经历的状态与专家的动作对（即从专家得到的动作建议）加入到原有的数据集中，这样新扩展的数据集既包含了专家演示中的状态，也包含了智能体在实际交互中遇到的状态
4. 策略更新：用新数据集训练得到下一个版本的策略，有效利用了专家反馈形成了更优策略
5. 迭代：本实验中，迭代的停止条件是根据超参数`num_updates`的次数来设置的，并未明确描述策略收敛的判断标准

通过本次实验，我可以感受到Dagger算法明显的**优缺点**：

- 减少累计误差：使得策略可以适应更广泛的状态分布，减少了在不熟悉状态下发生错误并导致错误积累的可能性
- 策略鲁棒性、动态调整的能力
- 每一轮迭代都需要专家标注反馈，成本高，根本标不起
- 策略初始化依赖专家，可能存在数据不足或质量不高