Assignment 1: Behavioral Cloning

In this assignment you will implement the DAgger algorithm for behavioral cloning (BC) [1]. The template code is available at https://github.com/xbpeng/rl_assignments. Installation instructions are provided in `README.md`. All of the files that need to be changed in this assignment are located in the `a1/` directory. Locations where code needs to be modified are labeled `TODO`.

The objective of behavioral cloning is to train a policy π to imitate the behaviors of an expert policy π^*. To run BC on a pre-trained policy for the `cheetah` task, use the following command:

```
python run.py --mode train \
--env_config data/envs/dm_cheetah.yaml \
--agent_config a1/dm_cheetah_bc_agent.yaml \
--log_file output/log.txt \
--out_model_file output/model.pt \
--max_samples 20000 \
--visualize
```

- `--env_config`: Configuration file for the environment.
- `--agent_config`: Configuration file for the agent, which contains hyperparameters for the agents.
- `--log_file`: A text files, which will record statistics from training, such as losses and performance of the model.
- `--out_model_file`: A checkpoint file containing the parameters of the trained model. During training, the latest model will be periodically saved to this file.
- `--max_samples`: Is the maximum number of samples/time steps the algorithm should collect for training the policy.
- `--visualize`: A flag to visualize the simulation. Visualization is mainly used for debugging and qualitative evaluation of the model's behaviors. Visualization should be disabled for faster training.

In the agent configuration file, `expert_model_file` specifies a checkpoint for the expert model that the BC agent will be trying to imitate.

During training, a tensorboard log `events.out.tfevents.*` will also be saved to the output directory. This can be used to monitor progress during training. To log can be viewed using tensorboard with the following command:

```
tensorboard --logdir=output/ --port=6006 --bind_all
```

Once a model has been trained, you can load a checkpoint and test the model with the following command:

```
python run.py --mode test \
--env_config data/envs/dm_cheetah.yaml \
--agent_config a1/dm_cheetah_bc_agent.yaml \
--model_file output/model.pt \
--visualize
```

- `--model_file`: A checkpoint file containing the parameters of the model.

1 DAgger

In the DAgger algorithm [1], a policy π is trained to imitate an expert policy π^* by optimizing the following objective:

$$\arg\min_{\pi} \; \mathbb{E}_{\mathbf{s}\sim p(\mathbf{s}|\pi)} \mathbb{E}_{\mathbf{a}^*\sim\pi^*(\mathbf{a}^*|\mathbf{s})} \left[-\log \pi(\mathbf{a}^*|\mathbf{s})\right], \tag{1}$$

where $\mathbf{s} \sim p(\mathbf{s}|\pi)$ denotes the state distribution induced by executing the policy π that is being trained. At each state $\mathbf{s}$, the agent queries the expert policy π^* to determine the action $\mathbf{a}^*$ that the expert would have taken at that state. π is then trained to imitate π^* by minimizing the negative log-likelihood of the expert's actions.

1.1 Action Inference

In `a1/bc_agent.py`, implement the `_decide_action(obs, info)` method, which is used by the agent to samples actions $\mathbf{a}$ from the policy π, and query the expert π^* for its actions $\mathbf{a}^*$. **Note:** the state $\mathbf{s}$ and the observation $\mathbf{o}$ will be used interchangeably in this assignment.

a) Given the current observations `obs`, sample an action from the policy $\pi(\mathbf{a}|\mathbf{s})$. `self._model` contains the model used to represent the policy. The model has a method `eval_actor(norm_obs)`, which returns an action distribution given a normalized observation `norm_obs`. The input observations `obs` to `_decide_action` represents the original unnormalized observations from the environment. However, different features can have drastically different scales, similarly for the actions. Therefore, in practice, it is often more effective to train the model to take normalized observations as input and then output normalized actions. `self._obs_norm` and `self._a_norm` are normalizers, which can be used to normalize and unnormalize observations and actions. `self._obs_norm.normalize(obs)` can be used normalize an observation.

Once you have the action distribution `a_dist` from the policy, you can then sample an action from this distribution via the method `a_dist.sample()`. Note, this action will be normalized. To unnormalize an action, use `self._a_norm.unnormalize(norm_a)`. The

unnormalize action from π should then be stored in the output variable `a`.

b) Next, query the expert policy $\pi(\mathbf{a}|\mathbf{s})$ for the expert action. The method `_eval_expert(obs)` can be used to to query the expert model for the action that the expert would have taken at a given state. In this case, the input should be an *unnormalized* observation, and the output of the method will be an *unnormalized* action. Store the expert's action in the output variable `expert_a`.

1.2 Loss Computation

In `a1/bc_agent.py`, implement the `_compute_actor_loss(norm_obs, norm_expert_a)` method, which is used to compute the loss for training the policy. The input `norm_obs` is a batch of *normalized* observations, and `norm_expert_a` is a batch of *normalized* actions recorded from the expert. Calculate the mean negative log-likelihood $-\log \pi(\mathbf{a}^*|\mathbf{s})$ across the batch, and store this values as a scalar tensor in the output variable `loss`. The action distribution from the policy has a method `log_prob(a)`, which can be used to calculate the log-likelihoods of actions.

1.3 Tasks

Train BC policies for the `cheetah` and the `walker` tasks using the env config files `data/envs/dm_cheetah.yaml` and `data/envs/dm_walker.yaml`. The corresponding agent config files for these tasks are `a1/dm_cheetah_bc_agent.yaml` and `a1/dm_walker_bc_agent.yaml`. Train each model for at least 20,000 samples, this can be specified in using the argument `--max_samples 20000`. Tune the hyperparameters in the agent config files `a1/dm_cheetah_bc_agent.yaml` and `a1/dm_walker_bc_agent.yaml`, so that the policies reach a return of at least 550 for `cheetah` and 200 for `walker`.

After training each model, plot a learning curve of the performance of the model at different training iterations. Use the plotting script `tools/plot_log/plot_log.py` to plot the data in the output log file `log.txt`. In `plot_log.py`, you can specify a list of files to plot in `files`. `x_key` and `y_key` specifies the variables for the x-axis and y-axis. `plot_title` specifies the title for the plot. Generate learning curves for the `cheetah` and `walker` training runs. The plots should use `Samples` for the x-axis, and `Test_Return` for the y-axis. Create separate plots for `cheetah` and `walker`, and specify the titles accordingly for each plot.

Submission

Your submission should contain the following files:

- `bc_agent.py`: code changes.

- `dm_cheetah_bc_agent.yaml`: tuned hyperparameters for the `cheetah`.
- `dm_walker_bc_agent.yaml`: tuned hyperparameters for the `walker`.
- `cheetah_bc_model.pt`: trained model for the `cheetah` task.
- `walker_bc_model.pt`: trained model for the `walker` task.
- `cheetah_bc_log.txt`: training log for the `cheetah` task.
- `walker_bc_log.txt`: training log for the `walker` task.
- `cheetah_bc_curve.png`: image of the learning curve for the `cheetah` task.
- `walker_bc_curve.png`: image of the learning curve for the `walker` task.

References

[1] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 627–635, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.