CSE 579-AU24: Intelligent Control through Learning and Optimization
Homework 1 - Supervised Learning of Behaviors

Due Wednesday October 23rd @ 11:59pm PST

The goal of this homework is to get an understanding of imitation learning, compounding error, and multi-modality. In particular, you will be implementing the behavior cloning and DAgger algorithms with both a Gaussian policy (unimodal) and an autoregressive model (multimodal). Please refer to the git link for the assignment https://github.com/WEIRDLabUW/CSE-579-HW1

Related Lectures: This homework largely involves knowledge from Lecture 2, 3 & 4 on Supervised Learning of Behaviors.

Collaboration: Students can discuss questions, but each student MUST write up their own solution, and code their own solution. We will be checking code/PDFs for plagiarism.

Late Policy: This assignment may be handed in up to 5 days late. If you have used up your 10 late days this quarter, there will be a penalty of 10% of the maximum grade per day.

1 Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib as well as Pytorch. If you are new to Pytorch, please refer to the following tutorial [Link]. The README describes how to install packages in a conda environment to solve this assignment. We also provide a link to a Google Colab notebook below that can be used for cloud execution. We recommend either using a Linux machine or using the Colab, rather than Windows. This section gives a brief overview and the README provides detailed instructions.

- `main.py` - overall launcher with hyperparameters and environment creation [ONLY MODIFY WHEN ASKED FOR PARAMETER TUNING]
- `environment.yml` - Conda env file to install dependencies [DO NOT MODIFY]
- `utils.py` - functions for data collections and policies. [TODOs in the class PolicyAutoRegressiveModel]
- `evaluate.py` - Evaluating learned policy reward and success rate [DO NOT MODIFY]
- `bc.py` - Code for behavior cloning [TODOs]
- `dagger.py` - Code for DAgger [TODOs]
- `DiffusionPolicy.py` - Code for extra credit diffusion [TODOs].
- `Colab Notebook` - Notebook for colab if you do not have access to a Linux machine at https://colab.research.google.com/drive/1FXA4XwaYmsZc4Sxt6fvD6FMqNBhngfUE?usp=sharing

Environment Details You are provided with two simulation environments: Reacher-a 2D environment where a double-jointed arm aims to move its end effector to a target location, and PointMaze-a 2 DoF ball that is

force-actuated in the cartesian directions x and y, to reach a target goal in a closed maze. To know more about the observation space and action space check out [Reacher] [PointMaze]

To get started, you must install the correct dependencies for the code to run. There are several ways to do this - if you have a Linux machine, install a conda environment (install Anaconda from here https://docs.anaconda.com/free/anaconda/install/index.html) with the correct dependencies from the provided environment.yml file using the following command

```
conda env create -f environment.yml
conda activate cse579a1
```

Then follow the instructions in the readme to install mujoco and other last dependencies. This should give you an environment with all the necessary dependencies, and you can run the code by running `main.py` as described in each section below.

If you do not have a Linux machine, it is advisable for you to use Google Colab instead. This will provide you with a single GPU instance that should be sufficient to run this code. The Colab instance can be found https://colab.research.google.com/drive/1FXA4XwaYmsZc4Sxt6fvD6FMqNBhngfUE?usp=sharing, and all the installs are completed by just running the initial cells inline. Please come talk to the course staff if you are having environment setup issues!

2 Behavior Cloning [20 points]

In this part, your task is to implement behavior cloning with a Gaussian policy and evaluate your model in both Reacher and PointMaze environments.

```
$ python main.py --env reacher --train behavior_cloning --policy gaussian
$ python main.py --env pointmaze --train behavior_cloning --policy gaussian
```

In this part, you are implementing behavior cloning with a Gaussian Policy. We've already provided implementation on the Gaussian Policy in `utils.py`. While you do not have to write this, take a look, it will help you learn about how these policies are implemented. In this example, you are given an expert dataset from a pre-computed expert policy that we have provided. Given this expert dataset, your goal is to implement maximum likelihood - sample a batch of actions and learn the parameters θ that maximize the log likelihood of the batch of actions a under the policy distributions $\pi_\theta(\cdot \mid s)$ inferred by pushing the batch of states s forward through the policy π_θ. This can be done by using the `log_prob` function provided in the Gaussian policy class to construct an appropriate loss function. Please make sure to convert to the appropriate PyTorch commands. chFor this task, you can anticipate that you will achieve a success rate exceeding 0.1, it's not strictly necessary for every run to surpass this threshold. But the average over several runs should provide you success over 0.1.

2.1 Expected Results

We provide an example below which serves as a reference for the expected output; please note that the success rate and reward may fluctuate:

```
$ python main.py --env reacher --train behavior_cloning --policy gaussian
using device cuda
Imported Expert data successfully
[0] loss: 1.82484010
```

```
[1] loss: 1.75394077
[2] loss: 1.67324485
[3] loss: 1.58506186
[4] loss: 1.47073979
[5] loss: 1.37155325
[6] loss: 1.31822487
[7] loss: 1.22702936
[8] loss: 1.06962404
[9] loss: 0.96061741

...
Success rate:  0.44
Average reward (success only):  -14.292164424317901
Average reward (all):  -19.48123450337574
```

2.2 Execution

1. Fill in the blanks in the code marked with TODO in the `simulate_policy_bc` function in `bc.py`.
2. Plot the loss during the training with default hyper-parameters. Report the success rate and average reward using the evaluate function that is provided to you.
3. Experiment with one set of hyperparameters that affects the performance of the behavioral cloning agent, such as the amount of training steps, the amount of expert data provided, or something that you come up with yourself. For one of the tasks used in the previous question, show a graph of how the BC agent's performance varies with the value of this hyperparameter. In the caption for the graph, state the hyperparameter and a brief rationale for why you chose it.

3 DAgger [30 points]

In this part, your task is to implement DAgger with a Gaussian policy and evaluate your model in both Reacher and PointMaze environments.

```
$ python main.py --env reacher --train dagger --policy gaussian
$ python main.py --env pointmaze --train dagger --policy gaussian
```

Please make sure to convert to the appropriate PyTorch commands. Remember that the key idea behind DAgger is to perform behavior cloning iteratively, growing the dataset by rolling out a behavior cloned policy, relabeling actions at visited states using an "interactive expert" (in your case, this will be an oracle expert policy that we provide to you to provide expert labels) and constantly accumulating further states and actions in the dataset. Doing so, you are expected to get a success rate higher than 0.8 on the Reacher and PointMaze environments.

3.1 Expected Results

When you execute the code, you may get similar outputs as below. Please note that the success rate and reward may fluctuate. Please note that training DAgger will take longer compared to Behavior Cloning, snice this will iteratively train on a growing dataset.

```
$ python main.py --env reacher --train dagger --policy gaussian
```

```
using device cuda
Imported Expert data successfully
Expert policy loaded
Expert policy loaded
[1,     7] loss: 1.50206782
[2,     7] loss: 0.56514988
[3,     7] loss: -0.58795891
[4,     7] loss: 0.49312799
[5,     7] loss: 0.68755144
[6,     7] loss: -0.11015398
[7,     7] loss: -0.43034067
[8,     7] loss: -0.42246126
[9,     7] loss: -0.65084269

...
Success rate:  0.99
Average reward (success only):  -4.38895142781317
Average reward (all):  -4.442805346502077
```

3.2 Execution

1. Fill in the blanks in the code marked with TODO in the `simulate_policy_dagger` function in `dagger.py`.
2. Plot the loss during the training with default hyper-parameters. Report the success rate and average reward using the evaluate function that is provided to you.
3. Compare the success rate and reward of the DAgger policy with the behavior cloning policy and explain why DAgger performs better.
4. Experiment with one set of hyperparameters that affects the performance of the agent, such as the amount of training steps, the amount of expert data provided, or something that you come up with yourself. For one of the tasks used in the previous question, show a graph of how the agent's performance varies with the value of this hyperparameter. In the caption for the graph, state the hyperparameter and a brief rationale for why you chose it.

4 Autoregressive Model [50 points]

Recall that in the class, we've learned that policy expressivity is a combination of expressivity of the function approximator and of the distribution family. So far, you've implemented BC and Dagger using a Gaussian policy. In this part, you will be implementing an autoregressive policy and comparing your results with the Gaussian policy. Below, we provide a recap of the key idea behind an autoregressive policy

Autoregressive policies: The key idea behind autoregressive policies is to leverage the power of categorical distributions for multimodality, but to make this applicable over multidimensional, continuous action spaces. Categorical distributions are naturally able to capture multimodality, but we need to map from continuous to discrete action spaces. For a one dimensional action space, this can be done by *discretizing* the action space to a set of discrete "buckets", and then converting the problem to one of classifying which bucket the discretized action will fall into. The challenge of discretization over action dimensions greater than 1, is that the number of buckets grows exponentially. So let's say every dimension has B buckets, and there are D dimensions, a fully discretized system will have B^D buckets. This quickly becomes impractical.

Autoregressive discretization uses per-dimension categorical discretization incurring linear rather than exponential number of buckets (BD vs B^D). The key idea to do so is leveraging the chain rule of probability. For a multidimensional action space $a = (a_1, a_2, a_3, \ldots, a_n)$, and a state s, the likelihood can be decomposed as follows:

$$p(a|s) = p(a_1, a_2, \ldots, a_n|s) \tag{1}$$
$$= p(a_1|s)p(a_2|a_1, s)p(a_3|a_2, a_1, s)\ldots p(a_n|a_{n-1}, \ldots, a_1, s) \tag{2}$$

and accordingly, the log likelihood is

$$\log p(a|s) = \sum_{i=1}^{n} \log p(a_i|a_{i-1}, a_{i-2}, \ldots, s) \tag{3}$$

Now each term $\log p(a_i|a_{i-1}, a_{i-2}, \ldots, s)$ at some dimension i can be represented by a neural network that inputs the current state s, the previous actions $(a_1, a_2, \ldots, a_{i-1})$ and predicts a categorical distribution over the action at the i'th dimension a_i. This allows us to retain multimodality, while preventing exponential memory and compute.

This model can be trained with standard maximum likelihood updates on this decomposed log probability. The important thing to remember is that at *training* time, you must use the true (discretized) actions $a_{i-1}, a_{i-2}, \ldots, a_1$ as input, whereas during policy execution/inference, you must sample dimensions one by one, using the previously sampled actions as conditioning for the next.

Fill in TODOs in `PolicyAutoRegressive` class inside `utils.py`, and run your behavior cloning and DAgger using your autoregressive model.

4.1 Execution

1. Fill in the blanks in the code marked with TODO in the `PolicyAutoRegressiveModel` class in `utils.py`.
2. Plot the loss during the training using behavior cloning and DAgger, with default hyper-parameters for the pointmaze environment. Report the success rate and average reward using the evaluate function that is provided to you.
3. Compare the success rate and reward of your autoregressive policies with training with Gaussian policies, and explain your observation.

```
$ python main.py --policy autoregressive --env pointmaze --train behavior_cloning

using device cuda
Imported Expert data successfully
[0] loss: 1.56780493
[1] loss: 1.02868288
[2] loss: 0.87738596
[3] loss: 0.79618543
[4] loss: 0.74271558
[5] loss: 0.70552745
[6] loss: 0.67830284
[7] loss: 0.65692271
```

```
[8] loss: 0.63891958
[9] loss: 0.62391058
...
Success rate:  0.84
Average reward (success only):  1.0
Average reward (all):  0.84

$
```

5 Diffusion Policies [Extra Credit, 50 points]

In this part, your task for extra credit is to implement diffusion policies for the point maze environment. To evaluate once implemented, run:

```
$ python main.py --env pointmaze --train diffusion --policy diffusion
```

5.1 Execution

1. For this, fill in the blanks in `DiffusionPolicy.py`. Some of the code relies on the Hugging Face diffusers library https://huggingface.co/docs/diffusers/en/index, which might be helpful to reference. The two blanks you need to implement is the model training, and the model sampling. The two functions to be completed are `get_action` and `train_diffusion_policy`. The `get_action` function is mean to do sampling/inference, while the `train_diffusion_policy` function trains the noise predictor.
2. Report the success rate on the point maze environment

Diffusion policies: The core concept of diffusion policies is to model the action distribution as a gradual denoising process, leveraging the power of score-based generative models. This approach allows for naturally capturing multimodal action distributions in continuous, high-dimensional action spaces without discretization. The key steps in a diffusion policy are:

Training Process: The training process starts by randomly drawing unmodified examples, $\mathbf{x}^0$, from the dataset. For each sample, we randomly select a denoising iteration k and then sample a random noise ϵ^k with appropriate variance for iteration k. The noise prediction network ϵ_θ is asked to predict the noise from the data sample with noise added.

$$\mathcal{L} = MSE(\epsilon^k, \epsilon_\theta(\mathbf{x}^0 + \epsilon^k, k)) \tag{4}$$

Thre resulting noise prediction network ϵ_θ can then be used to generate actions as below. This code is represented in the `train_diffusion_policy` function.

Sampling: The output generation process in diffusion models is modeled as a denoising process, often called Stochastic Langevin Dynamics (SGLD). Starting from $\mathbf{x}^K$ sampled from Gaussian noise, the diffusion model performs K iterations of denoising to produce a series of intermediate outputs with decreasing levels of noise, $\mathbf{x}^K, \mathbf{x}^{K-1}, ..., \mathbf{x}^0$, until a desired noise-free output $\mathbf{x}^0$ is formed. This sampling process follows the equation

$$\mathbf{x}^{k-1} = \alpha(\mathbf{x}^k - \gamma\epsilon_\theta(\mathbf{x}^k, k) + \mathcal{N}(0, \sigma^2 I)), \tag{5}$$

where ϵ_θ is the noise prediction network with parameters θ that will be optimized through learning (as described above) and $\mathcal{N}(0, \sigma^2 I)$ is Gaussian noise added at each iteration. This can be thought of as "noisy" gradient descent with ϵ_θ as the gradient. The relevant code is in the `get_action` function.

To improve the performance, the diffusion model is conditioned on a fixed number of previous states (the obs_horizon in the code). This is called observation stacking and helps reduce cycles. It also simultaneously predicts multiple actions in the future, a procedure referred to as action chunking. This also helps reduce cycles and speeds up inference as diffusion models are much slower. It predicts 12 actions into the future, executes the first 8 of them and then repeats prediction. There are more detailed instructions inline in the code with respect to low-level implementation. Given this is extra-credit, we also recommend reading the diffusion policies paper in detail for an in-depth description of necessary implementation details.

6 Submission

We will be using Canvas to submit the assignments. Please submit the written assignment answers as PDFs. For the code, submit a zip file of the entire working directory.