CSE 579-SP26: Intelligent Control through Learning and Optimization

Homework 1 - Supervised Learning of Behaviors

Due April 29th @ 11:59pm PST

The goal of this homework is to get an understanding of imitation learning, compounding error, and multimodality. 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/github/WEIRDLabUW/CSE-579-HW1/blob/main/CSE_579_HW1.ipynb

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/github/WEIRDLabUW/CSE-579-HW1/blob/main/CSE_579_HW1.ipynb, 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. For 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, since 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:

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

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 hyperparameters 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 a diffusion policy based on *denoising diffusion probabilistic models* (DDPM) on the PointMaze environment. You will write the noise scheduler, the forward and reverse processes, the training objective, and the sampling loop. To evaluate once implemented, run:

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

5.1 Background: Denoising Diffusion Probabilistic Models

A diffusion model defines a generative process by inverting a fixed Markovian *forward process* that gradually corrupts a clean sample into pure Gaussian noise over K discrete timesteps. The model learns the *reverse process*: starting from noise, iteratively denoise back to a clean sample. When this is applied to action sequences conditioned on observations, the result is a *diffusion policy* (Chi et al., 2023). Diffusion policies have become popular alternatives to Gaussian policies as they tend to be more expressive and provide better performance.

In what follows, we use x_0 to denote a clean (un-noised) action chunk drawn from the expert dataset, x_k to denote the same chunk after k steps of forward noising, and x_K to denote pure Gaussian noise. The conditioning on the observation history s is implicit throughout (we suppress it from the notation for readability), and the noise prediction network is denoted $\epsilon_\theta(x_k, k, s)$.

Forward (noising) process

The forward process is a fixed Markov chain that adds Gaussian noise according to a variance schedule $\beta_1, \beta_2, \ldots, \beta_K$:

$$q(x_k \mid x_{k-1}) = \mathcal{N}\Big(x_k;\ \sqrt{1-\beta_k}\, x_{k-1},\ \beta_k I\Big). \tag{4}$$

A key property of this process is that it admits a closed form for x_k given x_0. Define $\alpha_k = 1 - \beta_k$ and the cumulative product $\bar{\alpha}_k = \prod_{i=1}^{k} \alpha_i$. Then

$$q(x_k \mid x_0) = \mathcal{N}\big(x_k;\ \sqrt{\bar{\alpha}_k}\, x_0,\ (1 - \bar{\alpha}_k) I\big)\,, \tag{5}$$

which lets us sample x_k in one shot, without iterating through k steps:

$$x_k = \sqrt{\bar{\alpha}_k}\, x_0 + \sqrt{1 - \bar{\alpha}_k}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I). \tag{6}$$

This is the equation you will implement in `NoiseScheduler.add_noise`.

Beta schedules

The choice of $\{\beta_k\}$ controls how quickly information is destroyed. You will implement a "linear" schedule. Originally proposed in DDPM (Ho et al., 2020), β_k are linearly spaced between a small β_{start} and a larger β_{end} (typical values 10^{-4} and 0.02).

Training objective

Ho et al. (2020) show that, with the parameterization in (6), the variational lower bound on $\log p_\theta(x_0)$ reduces (up to constants and reweighting) to a simple denoising regression objective. Concretely, the network ϵ_θ is trained to predict the noise that was added:

$$\mathcal{L}(\theta) = \mathbb{E}_{x_0,\, \epsilon,\, k}\Big[\, \big\| \epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_k}\, x_0 + \sqrt{1 - \bar{\alpha}_k}\, \epsilon,\ k) \big\|^2 \,\Big]\,, \tag{7}$$

with $\epsilon \sim \mathcal{N}(0, I)$ and $k \sim \text{Uniform}\{0, 1, \ldots, K-1\}$.

In pseudocode the inner training step is:

```
for each batch (x_0, conditioning) in the dataset:
    eps    ~ N(0, I)                    # same shape as x_0
    k      ~ Uniform{0, ..., K-1}       # one per batch element
    x_k    = sqrt(alpha_bar_k) * x_0 + sqrt(1 - alpha_bar_k) * eps
    eps_p  = eps_theta(x_k, k, conditioning)
    L      = MSE(eps_p, eps)
    Adam.step(L)
```

This is what you implement in `train_diffusion_policy`.

Reverse (denoising) process

To generate a sample, we start from $x_K \sim \mathcal{N}(0, I)$ and iteratively denoise. The learned reverse step models each transition as

$$p_\theta(x_{k-1} \mid x_k) = \mathcal{N}\big(x_{k-1};\ \mu_\theta(x_k, k),\ \sigma_k^2 I\big)\,. \tag{8}$$

Ho et al. show (their eq. 11) that, when the network predicts noise $\epsilon_\theta(x_k, k)$, the mean can be written directly as

$$\mu_\theta(x_k, k) = \frac{1}{\sqrt{\alpha_k}} \left(x_k - \frac{\beta_k}{\sqrt{1 - \bar{\alpha}_k}}\, \epsilon_\theta(x_k, k) \right). \tag{9}$$

For the variance we use the simpler of the two interchangeable choices from Ho et al. (their Section 3.2):

$$\sigma_k^2 = \beta_k. \tag{10}$$

A sample from p_θ is then

$$x_{k-1} = \mu_\theta(x_k, k) + \mathbf{1}_{k>0}\sqrt{\beta_k}\, z, \qquad z \sim \mathcal{N}(0, I). \tag{11}$$

The indicator is important: at the very last step ($k = 0$) we return the mean directly without adding noise.

In pseudocode the sampling loop is:

```
x_K ~ N(0, I)
for k = K-1, K-2, ..., 0:
    eps_pred = eps_theta(x_k, k, conditioning)
    mu       = (1/sqrt(alpha_k)) * (x_k - beta_k/sqrt(1-alpha_bar_k) * eps_pred)
    if k > 0:
        x_{k-1} = mu + sqrt(beta_k) * randn_like(x_k)
    else:
        x_{k-1} = mu
return x_0
```

This is what you implement in `NoiseScheduler.step` and the loop inside `DiffusionPolicy.get_action`.

Observation stacking and action chunking

The vanilla diffusion policy described above predicts a single action conditioned on a single observation. In practice, performance improves substantially with two simple modifications proposed by Chi et al. (2023):

Observation stacking. Rather than conditioning on a single observation s_t, condition on a fixed-length history $(s_{t-O+1}, \ldots, s_t)$ where O is the `obs_horizon`. This gives the policy short-term temporal context (e.g. velocity) without needing recurrence.

Action chunking. Rather than predicting a single action a_t, predict an entire chunk of H_p future actions $(a_t, a_{t+1}, \ldots, a_{t+H_p-1})$ in one diffusion sample, then *execute* only the first H_a of them before re-querying the policy. The diffusion model thus produces a temporally consistent action sequence rather than a sequence of i.i.d. marginals, reduces oscillations found in single-step models. In our setup we use `obs_horizon` $= 4$, `action_pred_horizon` $= 12$, and `action_horizon` $= 8$. The dataset class and the action-chunk slicing in `get_action` are provided – you do not need to re-derive them.

5.2 What you implement

You will fill in five `TODO` blocks inside `DiffusionPolicy.py`. The 1D UNet noise prediction architecture (`ConditionalUnet1D`) is provided in `policy.py`. The dataset, optimizer, learning-rate schedule, and observation/action-chunk plumbing are also provided.

1. `NoiseScheduler.__init__` – build the β, α, and $\bar{\alpha}$ tables for both the linear schedule.
2. `NoiseScheduler.add_noise` – the closed-form forward sample from (6).
3. `NoiseScheduler.step` – one reverse DDPM step using (9): compute the mean μ_θ, and sample x_{k-1} with variance $\sigma_k^2 = \beta_k$ (with the no-noise boundary case at $k = 0$).
4. `train_diffusion_policy` (inner loop) – sample ϵ, sample k, build x_k via the scheduler, predict the noise, and compute the MSE loss.
5. `DiffusionPolicy.get_action` – the sampling loop: initialize from $\mathcal{N}(0, I)$, denoise for K steps using the scheduler, unnormalize, and return the action chunk.

5.3 Execution

1. Fill in the five `TODO` blocks above. You may not import `diffusers` or any other diffusion library; the entire DDPM machinery must come from your own code plus PyTorch.

2. Train the policy on PointMaze with the default hyperparameters and report the success rate using the provided `evaluate` function. Feel free to play with the implementation!

6 Submission

We will be using Canvas to submit the reports, and Gradescope to submit the code. Please submit the written assignment answers as PDFs.

For the code, submit the following files (if you didn't use Colab):

1. `bc.py`

2. `dagger.py`

3. `utils.py`

4. `DiffusionPolicy.py`

If you used Colab, download the Python version of your Jupyter notebook via:
File → Download → Download .py. Upload the resulting `.py` file to the autograder.
Finally, upload the following checkpoints to gradescope:

1. `gaussian_reacher_behavior_cloning_final.pth`

2. `gaussian_reacher_dagger_final.pth`

3. `gaussian_pointmaze_behavior_cloning_final.pth`

4. `gaussian_pointmaze_dagger_final.pth`

5. `autoregressive_pointmaze_behavior_cloning_final.pth`

NOTE: Do not upload the extra credit trained diffusion checkpoint (`diffusion_pointmaze_diffusion_final.pth`), as it exceeds the Gradescope 100 MB max file size.