Homework 4

CMU 10-703: Deep Reinforcement Learning (Fall 2025)
OUT: Wednesday, October 1, 2025
DUE: Sunday, October 12, 2025 by 11:59pm ET

Instructions: START HERE

Note: This homework assignment requires a significant implementation effort. Please plan your time accordingly. General tips and suggestions are included in the 'Guidelines on Implementation' section at the end of this handout.

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 10 grace days for your homeworks. **For this assignment only, if you use a late day, you will be extended the entire fall break (due date would be Sunday, October 19). Note that you will at most use a single day for this HW.** Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy here for more information about grace days and late submissions: `https://cmudeeprl.github.io/703website_f25/`
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 4." Additionally, zip all the code folders into a directory titled `<andrew_id>.zip` and upload it the GradeScope assignment titled "Homework 4: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.
  - **Submitting GIFs:** Upload your gifs to a google drive folder and share the link that with viewing permissions for CMU emails.

[1]`https://www.cmu.edu/policies/`

Problem 0: Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

Problem 1: CMA-ES (24 pts)

In this problem you will implement CMA-ES, a black-box optimization algorithm. To help you get started, we have provided some template code in this Notebook:

`https://colab.research.google.com/drive/1jjRDAQ3l78CZTZ_FKZKLaABlb0rFgYsg?usp=sharing`

You are welcome to implement the assignment from scratch using whatever programming language you prefer.

1. [**10 pts**] Implement CMA-ES using the following update equations:

$$\mu_{t+1} \leftarrow \frac{1}{\text{elite size}} \sum_{i=1}^{\text{elite size}} \theta_t^{(i)}, \qquad \Sigma_t \leftarrow \text{Cov}\left(\theta_t^{(1)}, \cdots, \theta_t^{(\text{elite size})}\right) + \epsilon I,$$

   where $\theta_t^{(i)}$ denotes the i-th best parameters from the previous iteration and $\epsilon \in \mathbb{R}$ is a small constant. We recommend the following hyperparameters:

   - Initial μ: $\vec{0}$
   - Initial covariance: $100I$
   - Population size: 100
   - Fraction of population to keep at each iteration: 10%
   - Noise ϵ added to covariance at each step: $0.25I$

   Run your implementation of CMA-ES to *maximize* on the following simple objective function:

$$f(x) = -\|x - x^*\|_2^2 \quad \text{where} \quad x^* = [65, 49].$$

   This function is optimized when $x = x^*$. Run your implementation of CMA-ES on this function, confirming that you get the correct solution. Make a plot showing the values of μ from each iteration. Use $\mu[0]$ for the X axis and $\mu[1]$ for the Y axis. Please label the initial and final values of μ, as well as the global optimum. Remember to label your axes.

2. [**4 pts**] In the second part of this problem, you will use CMA-ES to solve a RL problem. You will use the `Cartpole-v0` environment from OpenAI gym. Our first task will be to make an objective function that takes as input the parameters of a policy and outputs the reward of that policy. We will parametrize the policy as

$$\pi(a = \text{LEFT} \mid s) = s \cdot w + b,$$

   where $w \in \mathbb{R}^4$ and $b \in \mathbb{R}$ are parameters that you will optimize with CMA-ES. Define a function that takes as input a single vector $x = (w, b)$ and the environment and returns the total (undiscounted) reward from one episode. To check your implementation, evaluate the following policies 1000 times and report the average total reward (we've provided the answer for the first policy):

$x = (-1, -1, -1, -1, -1)$	$x = (1, 0, 1, 0, 1)$	$x = (0, 1, 2, 3, 4)$
15.6		

3. [**10 pts**] Run CMA-ES on the RL objective function. CMA-ES should be able to get an average population reward of at least 195 in 10 iterations. Include plot showing mean sample reward and best sample reward (Y axis) across iterations (X axis). Remember to label both lines and both axes.

Problem 2: Imitation Learning (62 pts)

In this problem, you will implement imitation learning from demonstrations. We will be using the `BipedalWalker-v3` environment. The expert trajectories to imitate will be supplied by an algorithm, specifically the RL method of Proximal Policy Optimization (PPO) [7], instead of human experts. You will have access to both the trajectories for BC and the actual PPO policy for DAgger. The expert trajectories to use are `actions_BC.pkl` and `states_BC.pkl` and are stored at *this link*. The checkpoint for the expert policy is at `data/models/super_expert_PPO_model.pt`. Use the provided `environment.yaml` for compatibility.

2.1. Behaviour Cloning (14 pts)

First you will implement Behavior Cloning. In behavior cloning, we simply use regression to train a policy to map observations to actions, supervised by a set of expert trajectories, i.e., sequences of state and actions. For the sake of evaluation, we will also be monitoring the reward of the training and generated trajectory segments. Our policy architecture is a residual three layer multilayer perceptron, and outputs actions in the range [-1,1].

The reference solutions takes less than 5 minutes to train and generate visualizations for each (with a GPU).

1. Implement the `generate_trajectories`, `train`, and `run_training` functions:

   - Implement the `generate_trajectories` function, which simply runs inference for a given number of trajectories with self.model using the `generate_trajectory` method and returns the rewards.
   - Implement the `train` function to run training when self.mode is "BC". We have provided a function that computes gradients and updates the model already, called `training_step`. In `train`, you should loop for `num_batch_collection_steps` steps. For each batch collection step, perform `num_training_steps_per_batch` training steps using `training_step`. Note that since we provide the expert trajectories, you don't actually need to rollout the expert policy to get these trajectories, just sample them from the data. For evaluation, compute/store the

average, median, and max reward from the `num_trajectories_per_batch` trajectories every `num_training_steps_per_batch` training steps using the function implemented previously.

- For `run_training`, create a SimpleNet model with the following parameters:
  - hidden_layer_dimension: 128
  - max_episode_length: 1600

  Use the AdamW optimizer with the following parameters:
  - learning rate: 0.0001
  - weight decay: 0.0001
- Pass all necessary information into a `TrainDagger` object (with `mode = "BC"`), which can then be trained with the `TrainDagger.train()` method. Collect 20 batches, each with 20 trajectories per batch. Train for 1000 steps per batch with a batch size of 128.

2. [**6 pts**] Save the training loss plot, *and write down the final loss value.*

3. [**6 pts**] Create a graph of the average, median, and max reward for trajectories sampled with your trained policy. Report these statistics for 20 trajectories every 1000 training steps until training is concluded.

4. [**2 pts**] After training, create and save a gif of the walker during a failure run of behavior cloning (reward below zero). Save the .gif as `"gifs_BC.gif"` and add it as one of the files in the code submission.

2.2. DAgger (18 pts)

Next, you will implement the DAgger algorithm [6] in `train_dagger_BC.py`. We have provided pseudocode in Algorithms 1 and 2.

1. Implement the `update_training_data` function, and update the `run_training` and `train` functions:

   - `update_training_data` should follow Algorithm 2.
   - Update the `train()` function to follow Algorithm 1 when self.mode is DAgger where instead of simply generating trajectory using our model, we also query the expert policy.
   - Use the same hyperparameters as the BC with the same initial model but set `mode = "DAgger"`. Collect 20 batches, each with 20 trajectories per batch. Train for 1000 steps per batch with a batch size of 128.

2. [**6 pts**] Save the training loss plot (it may have some spikes), *and write down the final loss value.*

3. [**6 pts**] Create a graph of the average, median, and max reward for each batch_collection_step when collecting trajectories using the DAgger model, and plot the results.

4. [**2 pts**] After training, create and save a gif of the walker during a successful run of behavior cloning (reward above 260). Save the .gif as `"gifs_DAgger.gif"` and add it as one of the files in the code submission.

5. [**4 pts**] Which method worked better, BC or DAgger, and why?

Algorithm 1 DAgger
Policy π_θ, Expert policy π^*_ϕ, num_batch_collection_steps=n, num_steps_per_batch=m

```
1: procedure TRAIN
2:     for $i$ in range($n$):
3:         query_expert_policy($\pi_\theta, \pi^*_\phi$)
4:         for $j$ in range($m$):
5:             # gets a minibatch and does a gradient update of $\pi_\theta$
6:             training_step($\pi_\theta, \mathcal{D}$)
7: end procedure
```

Algorithm 2 Query expert policy
Policy π_θ, Expert policy π^*, num_trajectories=N, current dataset $\mathcal{D}$

```
1: procedure QUERY_EXPERT_POLICY
2:     for $i = 1$ to $N$ do
3:         Sample trajectory $\tau_i = [s_i^1, a_i^1, ..., s_i^T, a_i^T]$ by deploying policy $\pi_\theta$.
4:         Get expert labels on the states of $\tau_i$ : $\mathcal{D}_i = \{(s_i^t, \pi^*(s_i^t), t = 1..T\}$.
5:         Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
6: end procedure
```

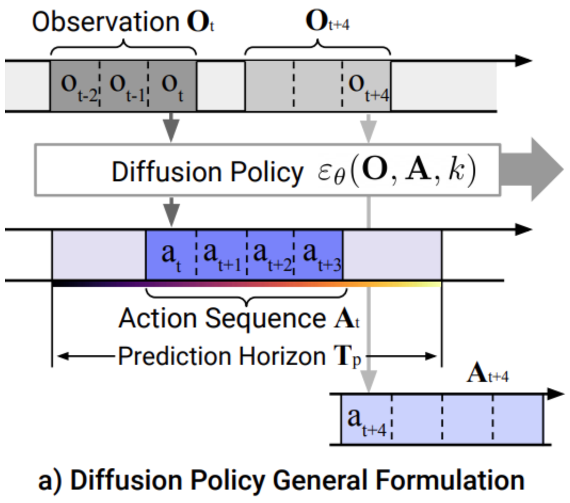
2.3. Diffusion policies [2, 3] (30 pts)

Next, you will implement learning from demonstrations with diffusion policies. Our policy architecture is transformer that takes as input a history of states and actions with their corresponding episode timesteps, noisy future actions and predicts the noise.

We provide a high level overview in Figure 1 as well as the architecture itself in `diffusion_policy_transformer.py`. If you have never seen a torch.nn.module implementation of a transformer network with cross attention/diffusion model, it may be interesting to look at `diffusion_policy_transformer.py`. Your implementation will be in `train_diffusion_transformer.py`.

The reference solution takes around 30 minutes to train and generate visualizations (with a GPU).

1. Implement the diffusion policy by updating the `run_training`, `training_step`, `diffusion_sample` and `sample_trajectory` methods of `train_diffusion_policy.py`.


Observation $\mathbf{O}_t$
$\mathbf{O}_{t+4}$
o_{t-2}
o_{t-1}
o_t
o_{t+4}
Diffusion Policy $\varepsilon_\theta(\mathbf{O}, \mathbf{A}, k)$
a_t
a_{t+1}
a_{t+2}
a_{t+3}
Action Sequence $\mathbf{A}_t$
Prediction Horizon $\mathbf{T}_p$
$\mathbf{A}_{t+4}$
a_{t+4}
a) Diffusion Policy General Formulation

(a) Inputs and outputs of the diffusion policy [2]. We take in previous observations (previous states and actions in our case), and output a set of future actions (3 actions in our case)

Figure 1

(a) Implement the `training_step` function in `train_diffusion_policy.py` via Algorithm 3. We will use Denoising Diffusion Probabilistic Models [4] with 30 denoising timesteps as our diffusion implementation, and this algorithm is a rough version of Algorithm 1 from the paper.

(b) For `run_training`, create a PolicyDiffusionTransformer model with the following parameters:

- 6 transformer layers
- hidden size 128
- 1 transformer head

Use the AdamW optimizer with the following parameters:

- learning rate: 0.00005
- weight decay: 0.001

Load the states, actions, and rewards using `pickle.load` from *this link*, and pass all necessary information into a `TrainDiffusionPolicy` object, which can then be trained with the `TrainDiffusionPolicy.train()` method. Train for 50000 steps with a batch size of 256. Save a model after 50000 steps.

(note) If your loss is below 0.01 or above 0.15 after 10000 iterations, your training is probably buggy - it is recommended to use wandb for viewing loss curves.

(c) [**6 pts**] Submit a training loss curve, and *write down the final loss value*

(d) Implement the `diffusion_sample` function as in algorithm 4

(e) Implement the `sample_trajectory` function using the diffusion transformer, as seen in Algorithm 5. Here are some helpful tips:

- During training, we normalized state values using self.states_mean and self.states_std before passing them into the model and normalized action values using self.actions_mean and self.actions_std before having the model predict them (this happened during initialization of `TrainDiffusionPolicy`). Make sure to account for this when running inference.
- During the beginning of inference, you will not have enough previous states/actions for full conditioning. Make sure to pad accordingly - padding should be False for states/actions to be included in the sequence, and True for states/actions to be padded, and padded states should be added to the end of the previous actions/previous states tensors (i.e. higher indices along the input_seq_length dimension, see the nn.module for diffusion_policy_transformer).

(**note**) for the following parts, feel free to update the `evaluation` method

(f) [**15 pts**] Generate 20 trajectories with `num_actions_to_eval_in_a_row=`1, 2, and 3, and calculate average time to run each process (This will take a long time to run. It is recommended to start with 3 actions evaluated in a row to test your setup, and the max value should be above the mean of the expert trajectories). What is the max, median, and mean reward in each case, and what is the average time to generate a trajectory in each case?

(g) [**2 pts**] After training, create and save a gif of the walker during a successful run of behavior cloning (reward above 240) with 3 actions evaluated in a row. Save the .gif and add it as one of the files in the code submission. Name the gif `"gifs_diffusion.gif"`, and add it to the code submission.

(h) [**4 pts**] Why does the diffusion policy takes so much longer to generate a trajectory than the simple model trained using DAgger or BC?

(i) [**3 pts**] Explain the difference in average time between your different test runs in part (f).

Feedback

Feedback: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing? **Feedback is especially useful for this homework, since this is the first time it has been used.**

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade.

Algorithm 3 Diffusion Policy Training Step
Denoising model ϵ_θ
NOTE: padding + episode timesteps removed for brevity

```
procedure TRAINING_STEP(batch_size)
    # our model conditions on $k$ input states, $k-1$ input actions,
    # and denoises $k'$ future actions
    $s_{i-k}, \dots, s_i, a_{i-k}, \dots, a_{i-1}, a_i, \dots, a_{i+k'}$ = get_training_batch(batch_size)
    $\epsilon \sim \mathcal{N}(0, I)$ # noise should be i.i.d across all actions/dimensions
                    # sample one $\epsilon$ per batch element
    $t \sim$ RandInt([1, T)) # sample one $t$ per batch element (i.i.d)
                              # use $T$ =self.num_train_diffusion_timesteps
    $[(a_i)_t, \dots, (a_{i+k'})_t] = \sqrt{\bar{\alpha}_t} * [a_i, \dots, a_{i+k'}] + \sqrt{1-\bar{\alpha}_t} * \epsilon$ # $(a_i)_t$ is $t$ noised action $a_i$
    # prev line implemented as:
    # noisy_actions = self.training_scheduler.add_noise(actions, $\epsilon$, $t$)
    $L(\theta) = \frac{1}{\text{batch\_size}} \|\epsilon_\theta(s_{i-k}, \dots, s_i, a_{i-k}, \dots, a_{i-1}, (a_i)_t, \dots, (a_{i+k'})_t, t) - \epsilon\|_2^2$
    # prev line implemented in torch as: loss = torch.nn.MSELoss()($\epsilon_\theta(\cdot), \epsilon$)
    Update weights $\theta$ with: AdamW($\nabla_\theta L(\theta)$)
end procedure
```

Algorithm 4 Sampling from Diffusion Policy
Denoising model ϵ_θ
NOTE: padding + episode timesteps removed for brevity

```
procedure DIFFUSION_SAMPLE(prev_actions, prev_states, num_guidance_steps=$N$)
    $\mathbf{x_T} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ # pure noise that will become actions
    $\mathbf{x_t} = \mathbf{x_T}$
    timesteps = get_inference_timesteps() # these will be in decreasing order $T, \dots, 1$
    for $t$ in timesteps:
        noise_levels = t
        model_inputs = prev_states, prev_actions, noisy_actions, noise_levels
        $z \sim \mathcal{N}(0, I)$ if $t > 1$ else $z = 0$
        $\mathbf{x_{t-1}} = \frac{\mathbf{1}}{\sqrt{\alpha_t}}\left(\mathbf{x_t} - \frac{\mathbf{1-\alpha_t}}{\sqrt{\mathbf{1-\bar{\alpha}_t}}}\epsilon_\theta(\text{model\_inputs})\right) + \sigma_\mathbf{t}\mathbf{z}$ # here we slightly denoise actions
        # prev 2 lines implemented as:
        # $\mathbf{x_t}$ = self.inference_scheduler.step($\epsilon_\theta$(model_inputs), noise_levels, $\mathbf{x_t}$)
    Return $\mathbf{x_0}$
end procedure
```

Algorithm 5 Sampling Trajectory
Denoising model ϵ_θ
NOTE: padding removed for brevity

```
procedure SAMPLE_TRAJECTORY(num_prev_states=$K$,num_actions_to_eval_in_a_row=$N$)
    $s, a, t, \text{done}, \text{truncated} = [\text{env.reset()}], [], [0], \text{False}, \text{False}$
    while not done and not truncated:
        # sample next actions
        actions = diffusion_sample($\epsilon_\theta, s, a, t$)
        for $i$ in range($N$):
            new_$s$, $r$, done, truncated, _ = env.step(actions[i])
            if done or truncated:
                return
            # append new tokens to sequence
            $s, a, t = s + [\text{new\_}s], a + [\text{actions[i]}], t + [\text{len}(s)]$
        # context length of transformer is at most $K$
        $s, a, t = s[-K :], a[-(K-1) :], t[-K :]$
end procedure
```

References

[1] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[2] Cheng Chi, Siyuan Feng, Yilun Du, Zhenjia Xu, Eric Cousineau, Benjamin Burchfiel, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *arXiv preprint arXiv:2303.04137*, 2023.

[3] Yiming Ding, Carlos Florensa, Pieter Abbeel, and Mariano Phielipp. Goal-conditioned imitation learning. In *Advances in Neural Information Processing Systems*, pages 15324–15335, 2019.

[4] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[5] Tim Pearce, Tabish Rashid, Anssi Kanervisto, Dave Bignell, Mingfei Sun, Raluca Georgescu, Sergio Valcarcel Macua, Shan Zheng Tan, Ida Momennejad, Katja Hofmann, et al. Imitating human behaviour with diffusion models. *arXiv preprint arXiv:2301.10677*, 2023.

[6] Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 1, page 6, 2011.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.