CS224R Spring 2026 Homework 1
Imitation Learning

Due 4/10/2026 9pm PT

SUNet ID:
Name:
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Overview

Goals: In this assignment you will experiment with imitation learning on a custom Flappy Bird environment using action chunking. You will train behavior cloning (BC) policies with different losses regression and flow matching, and improve it with DAgger.

1. Implement and train an imitation learning agent using regression.
2. Implement and train an imitation learning agent via flow matching to learn a generative policy.
3. Implement DAgger to improve the first regression policy via iterative relabeling with a deterministic expert.

Environment

The environment is a physics-based Flappy Bird where the agent controls a target y-position (normalised to $[0, 1]$). A PD controller converts the target into thrust internally, creating momentum-based dynamics.

- **Observation** (4-D, normalised): `[dist_to_pipe, gap1_y, gap2_y, bird_y]`.
- **Action**: target y-position in $[0, 1]$.
- **Easy mode**: one gap per pipe.
- **Hard mode**: alternating single and doulbe gap pipes.
- **Max episode length**: 1000 steps. An agent that survives all 1000 steps is considered successful.

Action chunking: Policies predict `ACTION_CHUNK=20` future target positions at once. During rollout, only the first `EXECUTE_STEPS=10` are executed before re-querying the policy (called receding horizon control).

Submission

Submitting the PDF: Make a PDF report containing the results to Problem 1, 2 and 3. And submit to gradescope on the **Homework 1 (Written Part)**.

Submitting the Code: Submit on the gradescope assignment called **Homework 1 (Programming Part)** a zip file containing:

- Your completed code files (`networks.py`, `losses.py`, `expert.py`, `dagger.py`) in the original format under the folder hw1 with TODOs filled in.
- The results from your runs `bc_reg_easy.txt`, `bc_reg_hard.txt`, `bc_flow_hard.txt`, `dagger_hard.txt`
- The resulting folder that you will submit should look like the following:

```
.
|-- hw1/                 # containing all the code
|-- bc_reg_easy.txt
|-- bc_reg_hard.txt
|-- bc_flow_hard.txt
|-- dagger_hard.txt
```

Use of AI Tools (e.g. ChatGPT, Cursor): For the sake of deeper understanding on implementing imitation learning methods, assistance from generative models to write code for this homework is prohibited. See the course website for more details.

Setup

See `installation.md` in the starter code for detailed instructions.
See `colab_instructions.md` if you prefer to run your code in google colab.

Codebase

The starter code lives in the `hw1/` directory. Your task is to fill in the functions marked with `TODO` (which raise `NotImplementedError`).

- `main.py` — training pipeline and CLI entrypoint (**read-only**)
- `visualization.py` — evaluation, video recording, policy wrappers (**read-only**)
- `flappy_bird_env.py` — Gymnasium environment (**read-only**)
- `expert.py` — Agent expert that provides the initial demos (**read-only**)
- `networks.py` — neural network architectures [**TODO:** `BCPolicy`, `FlowMatchingSchedule`]
- `losses.py` — loss functions [**TODO:** `bc_loss`, `flow_matching_loss`]
- `dagger.py` — DAgger relabeling [**TODO:** `DeterministicExpert.act`, `rollout_and_relabel`]

We recommend implementing in this order:

1. `networks.py::BCPolicy`
2. `losses.py::bc_loss`
3. `networks.py::FlowMatchingSchedule.interpolate` and `FlowMatchingSchedule.sample`
4. `losses.py::flow_matching_loss`
5. `dagger.py::DeterministicExpert.act`
6. `dagger.py::rollout_episode`
7. `dagger.py::rollout_and_relabel`

Problem 1: Behavior Cloning with Regression [2 points]

In this section, you will implement the Behavior Cloning (BC) algorithm to train your policies using the Mean Square Error (MSE) loss.

We provide a set of demonstrations of an expert playing our Stanford Flappy Bird game (see `expert.py`). This dataset $\mathcal{D}$ consists of state action pairs (s, a).

BC entails learning a policy π that matches from the state of the flappy bird environment s, to the action to be taken by the bird a.

Today, most robot learning policies predict the set of T future actions $a_t ... a_{t+T}$ out of which only $K < T$ are rolled out open loop (without re-querying the policy) and the rest are discarded. This technique, called action chunking, generally results in much better performance.

1. **Implementation.** Implement the following:

   - `BCPolicy.__init__` and `BCPolicy.forward` in `networks.py`: a 3-layer MLP (Linear $\rightarrow$ ReLU $\rightarrow$ Linear $\rightarrow$ ReLU $\rightarrow$ Linear $\rightarrow$ Sigmoid).
   - `bc_loss` in `losses.py`: MSE loss between predicted and expert actions:

   $$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} \|\hat{a}_i - a_i^*\|^2 \tag{1}$$

   where $\hat{a}_i = \pi_\theta(s_i)$ is the predicted action and a_i^* is the expert action.

2. **Run the regression agent on easy mode** and report your results in a table (mean and standard deviation of episode length over 50 evaluation episodes) [1 point].

   ```
   python main.py --method bc_reg --env easy
   ```

   Include your results table here.

3. **Run the regression agent on hard mode** and report the results in a table below (no new code needed):

   ```
   python main.py --method bc_reg --env hard
   ```

   Include your results table here.

4. **Explain** in 2–3 sentences the performance of the MSE regression on hard mode. Why is it performing the way it is? [1 point]

   Include your explanation here.

Problem 2: Flow Matching [2 points]

In this section, you will implement flow matching to train your policies instead of using an MSE loss. Flow matching learns a conditional vector field that iteratively transports noise into realistic action chunk predictions. Flow matching is similar to diffusion, but it is simpler to implement and generally exhibits similar or better performance.

Let $\mathbf{a}_t$ be an action chunk in the demonstration dataset and $\mathbf{a}_{t,0} \sim \mathcal{N}(0, I)$ be noise of the same shape. We first sample a "flow matching timestep" $\tau \sim \mathcal{U}(0, 1)$ and define the interpolation $\mathbf{a}_{t,\tau} = \tau \mathbf{a}_t + (1-\tau)\mathbf{a}_{t,0}$. We then train a network v_θ to predict the velocity that moves $\mathbf{a}_{t,\tau}$ toward $\mathbf{a}_t$, using the flow-matching loss:

$$\mathcal{L}_{\text{FM}}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(s_t, a_t) \in \mathcal{D}} \left\| v_\theta(\mathbf{s}_t,\, \mathbf{a}_{t,\tau},\, \tau) - (\mathbf{a}_t - \mathbf{a}_{t,0}) \right\|_2^2 . \tag{2}$$

At inference time, we sample initial noise $\mathbf{a}_{t,0} \sim \mathcal{N}(0, I)$ and integrate the ODE $\frac{d\mathbf{a}_{t,\tau}}{d\tau} = v_\theta(\mathbf{s}_t, \mathbf{a}_{t,\tau}, \tau)$ from $\tau = 0$ to $\tau = 1$. The simplest integration method is Euler integration, which is given by the following update rule:

$$\mathbf{a}_{t,\tau+\frac{1}{n}} = \mathbf{a}_{t,\tau} + \frac{1}{n} \cdot v_\theta(\mathbf{s}_t, \mathbf{a}_{t,\tau}, \tau), \tag{3}$$

which is repeated n times from $\tau = 0$ to $\tau = 1$ to obtain $\mathbf{a}_{t,1}$, where n is the number of integration steps (also called "denoising steps"). $\mathbf{a}_{t,1} = \mathbf{a}_t$ is the final action chunk that is executed as before.

1. **Implementation.** For this part, you will implement the `FlowMatchingSchedule` class in `networks.py` and `flow_matching_loss` in `losses.py`. Specifically, you should implement the following:

   - `FlowMatchingSchedule.interpolate`: given a clean action chunk $\mathbf{a}_1$ and timestep $\tau \in [0, 1]$, sample noise and return the interpolated $\mathbf{a}_\tau$ and target velocity $v = a_1 - \epsilon$.
   - `FlowMatchingSchedule.sample`: starting from $\mathbf{a}_0 \sim \mathcal{N}(0, I)$, run Euler integration for `num_steps` steps and return the result clamped to $[0, 1]$.
   - `flow_matching_loss`: implement the flow matching loss presented above (tip: call `schedule.interpolate`)

2. **Run flow matching on hard mode** and report your results in a table (mean and standard deviation of episode length) [1 point]:

   ```
   python main.py --method bc_flow --env hard
   ```

   Include your results here.

3. **Explain** in 2–3 sentences why flow matching performs the way it does on hard mode. [1 point]

   Include your results and explanation here.

Problem 3: DAgger [2 points]

In this section, you will implement the DAgger algorithm to improve the behavioral cloning (BC) policy trained with the regression objective in Problem 1.

As discussed in class, DAgger works by iteratively rolling out the current learned policy, π, to collect states encountered under its own behavior, which we store in $\mathcal{D}_\pi$. For each visited state s, the policy produces an action $\pi(s)$, but instead of keeping that action, we query the expert and relabel the state with the expert action. This gives the dataset

$$\mathcal{D}_{DAgger} = \{(s, \pi_{expert}(s)) \mid s \sim \mathcal{D}_\pi\}.$$

Once this relabeled dataset has been collected, we perform imitation learning on the aggregated dataset $\mathcal{D} \bigcup \mathcal{D}_{DAgger}$ using the same regression objective from Problem 1. This process is typically repeated for multiple DAgger iterations.

DAgger helps mitigate distribution shift between the expert and the learned policy by visiting states that might be present in the policy rollouts but not in the demos, and acquiring expert supervision in those states.

1. **Implementation.** Implement the following in `dagger.py`:

   - `DeterministicExpert.act`: same logic as `Expert.act` in hard mode, but pick a strategy that will resolve the ambiguity found in the previous questions so that you make the MSE regression policy work.
   - `rollout_episode`: roll out the current policy, don't forget to reset the environment and to use the action chunks from the policy and return the state action pairs.
   - `rollout_and_relabel`: roll out the current policy using `rollout_episode`, use `DeterministicExpert` to relabel the episodes, and get the new action state training pairs.

2. **Run DAgger on hard mode** and report your results as a learning curve [1 point]:

   ```
   python main.py --method dagger --env hard
   ```

   Run dagger for 5 rounds (the default value). Plot the DAgger learning curve (round number on x-axis, mean episode length on y-axis, with error bars for standard deviation). Include the original regression performance as a horizontal line on the same plot.

   Include your DAgger learning curve plot here.

3. **Comparison.** Create a bar chart or table comparing the performance of all three methods on hard mode: Regression, Flow Matching, and DAgger (final round). [0.5 points]

```
python main.py --plot
```

Include the comparison plot and table here.

4. **Explain** in 3–4 sentences: why does DAgger improve over rounds? What role does the deterministic expert play? How does this DAgger's approach solve the challenges mentioned previously for the MSE regression policy? [0.5 points]

Include your explanation here.