Deep Reinforcement Learning
Assignment 3

2023 Spring

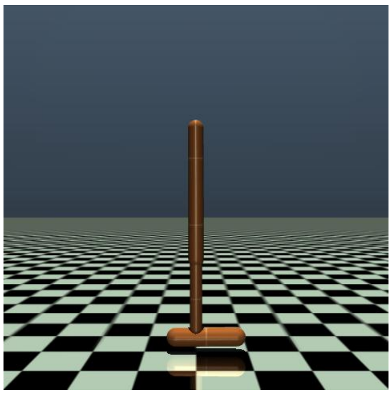
May 31, 2023

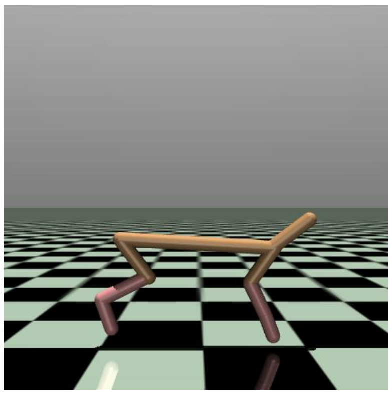
In this homework, you will implement the Decision Transformer [1] algorithm to solve three locomotion tasks of the MuJoCo environment using an offline dataset from D4RL [2].

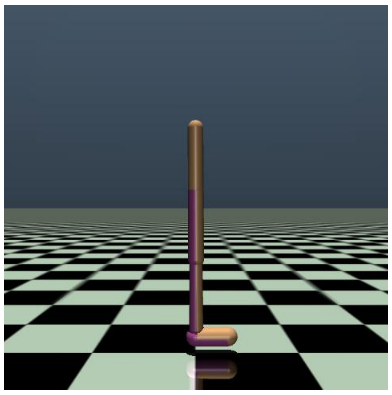
1 About the Environment

MuJoCo stands for Multi-Joint dynamics with Contact. It is a general-purpose physics engine that aims to facilitate research and development in robotics and other areas that demand fast and accurate simulation. Initially developed by Roboti LLC, it was acquired and made freely available by DeepMind in October 2021, and open-sourced in May 2022.

In this homework, we are using the gymnasium adaption of MuJoCo environments, an overview of which can be found here. Specifically, we are using the `Hopper-v4`, `Walker2d-v4`, and `HalfCheetah-v4` environments. The goal of these tasks is to train an agent to control the corresponding robot to move forward as fast as possible. The agent receives a reward proportional to the distance it travels in each step. The episode ends when the robot falls down or reaches a maximum number of steps.


Hopper


HalfCheetah


Walker2d

The state spaces for MuJoCo environments in Gymnasium consist of two parts that are flattened and concatenated together: a position of a body part ('mujoco-py.mjsim.qpos') or joint and its corresponding velocity ('mujoco-py.mjsim.qvel'). Often, some of the first positional elements are omitted from the state space since the reward is calculated based on their values,

Table 1: Mujoco Environments

Environment	Description	Obs. Dim.	Action Dim.	Target Return
Halfcheetah-v4	Bipedal Alternative Jumping	17	6	12 000
Hopper-v4	Monopedal Hopping	11	3	3 600
Walker2d-v4	Bipedal Alternative Walking	17	6	5 000

leaving it up to the algorithm to infer those hidden values indirectly. The action space is a vector of continuous values that represent the desired torque to be applied to each joint. Some details for the three environments are listed in Table 1.

2 About the Dataset

D4RL is an open-source benchmark for offline reinforcement learning. It provides standardized environments and datasets for training and benchmarking algorithms, and it is wildly used in offline RL methods. In this homework, we will use the D4RL dataset of the three environments to train our agent.

For each environment, D4RL provides three types of datasets, represented by `expert`, `medium`, and `medium-replay`. The `expert` dataset is collected by first training a policy using SAC, then collecting data by the full-trained agent. While the `medium` dataset is collected by a partially trained SAC agent, reaching around half the performance of the `expert` dataset. The `medium-replay` dataset is collected by the same partially trained SAC agent but with a different replay buffer. The `medium-replay` dataset consists of all samples in the replay buffer observed during training the SAC agent until reaching the "medium" performance.

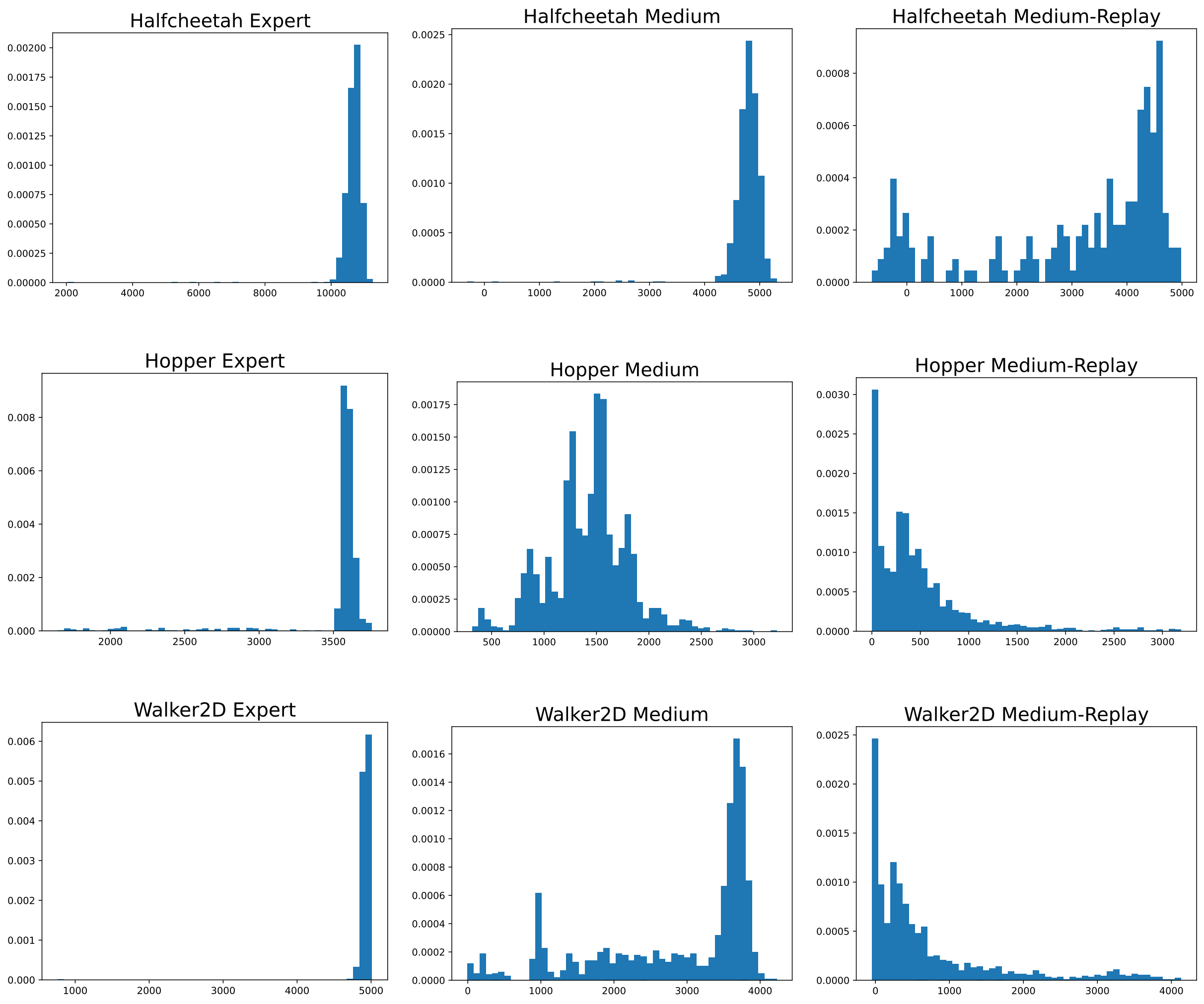
You can find the trajectory return distribution of the nine datasets of the three environments in Figure 1. For this homework, we only require you to provide results on the `medium` dataset. However, you are encouraged to try other datasets and compare the results.

3 About the Starting Code

Homework 5 follows the framework we've been working on starting from homework 2. You will find much of the code familiar. However, since offline RL doesn't need to interact with the environment and collect new data, we make some modifications in the `core.py`. Again, we are using the `hydra` framework to manage the configuration of the experiments. Please refer to `hydra`'s documentation for more details.

The results and saved files will be stored in the `runs` folder, under the subfolder specified by the time of execution. You can find the training curves and a video of the trained agent in the subfolder. If you want to turn off this behavior and save everything in the current folder, you can change the `hydra.run.chdir` field in the `config.yaml` file to `false`.

To improve the credibility of the results, we used three random seeds and plotted the current results' average and standard deviation periodically during training. We provide two versions of the program entry: `main.py` and `main_mp.py`. The former runs the experiments for the three seeds sequentially and updates the resulting standard deviation from the second seed on. This



Expert Datasets Medium Datasets Medium-Replay Datasets

Figure 1: Trajectory Return Distribution of D4RL Datasets

version is more suitable for debugging your code. The latter runs all three seed experiments in parallel and is suitable for quickly iterating and getting the results. You can choose either version to run, and after all three seeds, their results should be the same. You don't need to change anything in the program entries.

4 Requirements

Please submit a `zip` file containing the following files or folders:

- A code folder containing all the `.py` files and the `cfgs` folder.
- The auto-generated `runs` folder which contains the results for each of the required experiments and their corresponding config, log, and model files. You may remove the generated videos but leave the other files untouched. Please make sure the code you provide can reproduce the contents of the `runs` folder given the corresponding config file.

- A `report.pdf` file briefly summarizing your results. It should include the auto-generated results figure. You may also include any other findings you think are relevant.

You can find the reference training curves in the `gallery` folder. The TAs are using five seeds to generate results for the gallery, so we don't require your curves to be the same as ours anyway. However, if your results and losses are drastically different from the reference curves, there may be something wrong with your implementation.

We **only** require the results on the `medium` dataset, the required mean performance of each environment at the end of training (after 20_000 steps) are as follows:

- `Hopper Medium` The mean normalized d4rl-score should be greater than **50**
- `HalfCheetah Medium` The mean normalized d4rl-score should be greater than **40**
- `Walker2d Medium` The mean normalized d4rl-score should be greater than **70**

A success reproduce of the results for any dataset could give you **80pts**, while the rest 20pts will be given based on the performance of the other two datasets, **10pts** each.

Please refer to `README.md` for the dataset and environment instructions of this homework.

5 Decision Transformer [100 pts]

In this section, you will implement a simplified version of the Decision Transformer [1] agent which models an offline reinforcement learning problem into a sequential modeling problem. Compared to the original paper which uses the same architecture as GPT-2, we will use a simplified network from the repo minGPT while achieving similar performance in these tasks. We also modified the replay buffer we've been using to sample first trajectories and then transitions, maintaining the same behavior as the original paper while being (significantly) more efficient.

The Decision Transformer (DT) first maps the past K timesteps of rewards-to-go $g_{t-K:t}$, observations $s_{t-K:t}$, and actions $a_{t-K:t}$ to the same space of embedded tokens, then put these token into the transformer architecture sequentially and predicts the next tokens in the same way as in the Generative Pre-Training [3]. Let ϕ_g, ϕ_s and ϕ_a denote the reward-to-go, state, and action encoders respectively, the input tokens are obtained by first mapping the inputs to a d-dimensional embedding space, then adding a timestep embedding $\omega(t)$ to the tokens.

We use u_{g_t}, u_{s_t} and u_{a_t} to denote the input tokens corresponding to the reward-to-go, the observation, and the action of the timestep t respectively, and v_{g_t}, v_{s_t} and v_{a_t} be their counterparts on the output token side. DT could be formalized as:

$$\begin{gathered} u_{g_t} = \phi_g(g_t) + \omega(t), \quad u_{s_t} = \phi_s(s_t) + \omega(t), \quad u_{a_t} = \phi_a(a_t) + \omega(t) \\ v_{g_{t-K}}, v_{s_{t-K}}, v_{a_{t-K}}, \dots, v_{g_t}, v_{s_t}, v_{a_t} = \mathrm{DT}(u_{g_{t-K}}, u_{s_{t-K}}, u_{a_{t-K}}, \dots, u_{g_t}, u_{s_t}, u_{a_t}) \end{gathered} \tag{1}$$

DT uses a prediction head π_θ to get current action from the output of the state tokens, the training target of DT is to minimize the negative log-likelihood for the model to produce the real action in the dataset $\mathcal{T}$:

$$J(\theta) = \frac{1}{K}\mathbb{E}_{(\mathbf{a},\mathbf{s},\mathbf{g})\sim\mathcal{T}}\left[-\sum_{k=1}^{K} \log \pi_\theta\left(a_k \mid v_{s_k}\right)\right] \tag{2}$$

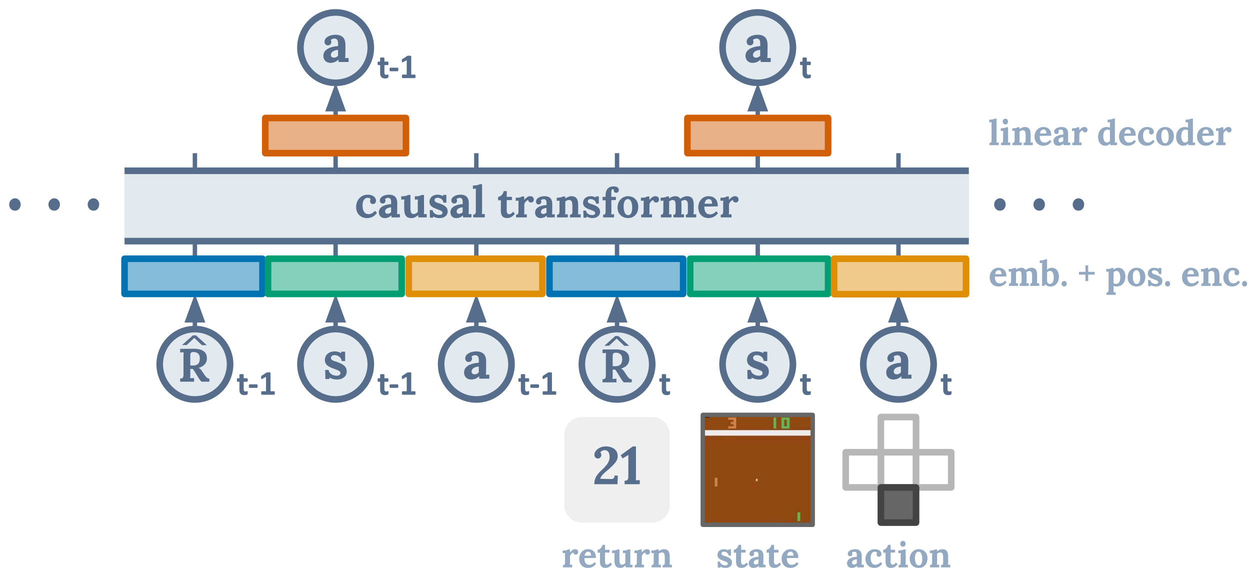


Figure 2: Architecture of the Decision Transformer

In this homework, you'll need to read through the following files and implement the missing parts:

`models.py` This file contains the model definitions for the `DecisionTransformer` class. Read through the `MaskedCasualAttention` class and `Block` class to get a sense of the architecture of a transformer, then implement the `forward` method of the `DecisionTransformer` class. First, fill in the part to get the input embeddings, then get the action predictions from the output tokens. You also need to implement the `_norm_state` and `_norm_action` methods, refer to the comments in the code for more details. Note how the shape of the input tokens and the hidden variable `h` are changed through the forward pass.

`buffer.py` This file contains the replay buffer classes we use to store D4RL trajectories and sample mini-batches from them. You need to complete the `__init__` method of the `SequenceBuffer` class, which fills the buffer with trajectories from the dataset. You can refer to the `ReplayBuffer` class for how to sample mini-batches from the buffer. A tool function `discounted_cumsum` is needed as well to compute the reward-to-go for each timestep in the trajectory from the immediate reward. You can read through the `sample_batch` method to get a sense of how the sampling is done from the 1D buffer.

`utils.py` This file implements a handful of tool functions we use in the training process. You don't need to implement anything in this file for this assignment.

`core.py` This file contains the main training and evaluation loop. This time, you'll need to fill the `eval` function since it's a bit more complicated for DT to first store past states, actions, and rewards-to-go, then predict an action from the current state. We also introduced a vectorized version of the `eval` function to speed up the evaluation process, because the model can get a batch of actions at each forward pass. You can also read through the `train` function to get a sense of how the training is done.

`main.py` This is the main file that you'll run to start the training process. You don't need to implement anything in this file for this section. There's also a multi-processing version of this file `main_mp.py` that you can use to train all seeds in parallel.

After implementing the necessary parts of the homework, you can run the following command to start the training process:

```
python main.py
```

We recommend you try out our parallel version of `main.py`:

```
python main_mp.py
```

where we use the `multiprocessing` library to train all seeds together in parallel. Feel free to read through the code of `main_mp.py` as well.

The default dataset we use is Hopper Medium, overriding the `env` and `buffer.dataset` variable to change to another dataset like Walker2d Medium-Replay

```
python main_mp.py env=walker2d buffer.dataset=medium-replay
```

6 Bonus [10 pts]

You can get 10 bonus points by implementing any of the following extensions, the total score will be capped at 100 points.

- The Decision Transformer paper claims their ability to adapt to different target rewards-to-go during test time, showing a near-linear curve of the real performance with respect to the target reward-to-go. You can try to test whether this is true for our minimized DT by running the eval function multiple times, changing the target reward-to-go in the process and plotting the reward curve to see if the performance changes accordingly. If so, please report your findings and the relative ablation results.
- Percentile Behavior cloning is a simple yet effective method to improve the performance of BC, which filters the dataset by trajectory return and only keeps the top $x\%$ trajectories. You can try to implement this method and see if it can improve the performance of DT in different datasets. If so, please report your findings and the relative ablation results.
- The authors of DT find that supervising the model with the next state and reward-to-go along with the action doesn't have much improvement over just using the action. You can try to verify this observation in different datasets. To this end, you'll need to calculate the `state_preds` and `reward_to_go_preds` in the `forward` function of the model and modify the `train` function to supervise on these predictions as well. Note that you'll need to shift the target states and rewards-to-go by one timestep to match the input tokens.
- If you think there's any bug in the code, please let us know and provide your solutions.

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] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning, 2020.

[3] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.