CS 4/5789 - Programming Assignment 4

April 15, 2024

This assignment is due on **May 3, 2024** at 11:59pm.

Section 0: Requirements

This assignment uses Python 3+. It is recommended to use Anaconda to install Python as well as the required libraries. The required libraries are

- `conda install -c conda-forge gym` (version 0.21.0 recommended)
- `conda install numpy` (version 1.22.0 recommended)
- `conda install -c conda-forge pyglet`
- `conda install -c conda-forge box2d-py`
- `conda install -c anaconda scikit-learn`
- `conda install -c conda-forge stable-baselines3`
- PyTorch[1]

Make sure not to add any additional imports. Some of the functions can be implemented with functions from SciPy but we want you to implement them yourself.

Section 1: Natural Policy Gradient

In this section, we will walk through Natural Policy Gradient (NPG) and also implement it for CartPole Simulation.

1.1 Background and Setting

We will consider the default CartPole simulator in OpenAI gym where we have two discrete actions $\mathcal{A} = \{0, 1\}$ (here 0 and 1 are the index of actions, and physically, 0 means applying a left push to the cart, and 1 means applying a right push to the cart. You just need to compute a stochastic policy which samples 0 or 1, and feed it to the step function which will do the rest of the job for you). Before defining the policy parameterization, let us first featurize our state-action pair. Specifically, we will use random fourier feature (RFF) $\phi(s, a) \in \mathbb{R}^d$, where d is the dimension of RFF feature. RFF is a randomized algorithm that takes the concatenation of (s, a) as input, outputs a vector $\phi(s, a) \in \mathbb{R}^d$, such that it approximates the RBF kernel, i.e., for any $(s, a), (s', a')$ pair, we have:

$$\lim_{d\to\infty} \langle \phi(s,a), \phi(s',a') \rangle = k\left([s,a],[s',a']\right),$$

where k is the RBF kernel on the concatenation of state-action (we denote $[s, a]$ as the vector $[s^\top, a]^\top$). In summary, RFF feature approximates RBF kernel, but allows us to operate in the primal space rather than the dual space where we need to compute and invert the Gram matrix (recall Kernel trick and Kernel methods from the ML introduction course) which is very computationally expensive and does not scale well to large datasets.

We parameterize our policy as follows:

$$\pi_\theta(a|s) = \frac{\exp\left(\theta^\top \phi(s,a)\right)}{\sum_{a'} \exp\left(\theta^\top \phi(s,a')\right)},$$

where the parameters $\theta \in \mathbb{R}^d$. Our goal is of course to find the best θ such that the resulting π_θ achieves large expected total reward.

[1]See `https://pytorch.org/get-started/locally/` for installation for local machine. Recommended to have torch 1.11.0.

1.2 Gradient of Policy's Log-likelihood

TODO Given (s, a), we need to first derive the expression $\nabla_\theta \ln \pi_\theta(a|s)$. This is part of the previous written homework. Now go to `utils.py` to implement the computation of $\nabla_\theta \ln \pi_\theta(a|s)$ in `compute_log_softmax_grad`. You can also implement `compute_softmax` and `compute_action_distribution` first to use them to calculate the gradient. You should pass the tests in `test.py` for these functions before moving on.

Remark The file `test.py` comprises of tests for some functions. If your implementation is correct, the printed errors of all tests should be quite small, usually not larger than 1e-6.

1.3 Fisher Information Matrix and Policy Gradient

We consider the finite-horizon MDP. Note that in class we considered the infinite-horizon version, but they are quite similar. Let us now compute the fisher information matrix. Let us consider a policy π_θ. Recall the definition of Fisher information matrix:

$$F_\theta = \mathbb{E}_{s,a \sim d_{\mu_0}^{\pi_\theta}} \left[\nabla_\theta \ln \pi_\theta(a|s) \left(\nabla_\theta \ln \pi_\theta(a|s) \right)^\top \right] \in \mathbb{R}^{d \times d}.$$

We approximate F_θ using trajectories sampled from π_θ. We first sample N trajectories $\tau^1, \ldots, \tau^N$ from π_θ, where $\tau^i = \{s_h^i, a_h^i, r_h^i\}_{h=0}^{H-1}$ with $s_0^i \sim \mu_0$. We approximate F_θ using all (s, a) pairs:

$$\widehat{F}_\theta = \frac{1}{N} \sum_{i=1}^{N} \left[\frac{1}{H} \sum_{h=0}^{H-1} \nabla_\theta \ln \pi_\theta(a_h^i|s_h^i) \nabla_\theta \ln \pi_\theta(a_h^i|s_h^i)^\top \right] + \lambda I,$$

where $\lambda \in \mathbb{R}^+$ is the regularization for forcing positive definiteness.

Remark Note the way we estimate the fisher information. Instead of doing the roll-in procedure we discussed in the class to get $s, a \sim d_{\mu_0}^{\pi_\theta}$ (this is the correct way to ensure the samples are i.i.d), we simply sample N trajectories, and then average over all state-action (s_h, a_h) pairs from all N trajectories. This way, we lose the i.i.d property (these state-action pairs are dependent), but we gain sample efficiency by using all data.

TODO First, go to `train.py` and finish the `sample` function to sample trajectories using the current policy. This function should rollout N trajectories and keep track of the gradients and rewards collected during each trajectory. Then go to file `utils.py` and implement $\widehat{F}_\theta$ in `compute_fisher_matrix`. Note that in OpenAI Gym CartPole, there is a termination criteria when the pole or the cart is too far away from the goal position (i.e., during execution, if the termination criteria is met or it hits the last time step H, the simulator will return *done = True* in step function. During generating a trajectory, it is possible that we will just terminate the trajectory early since we might meet the termination criteria before getting to the last time step H). Hence, we will see that when we collect trajectories, each trajectory might have different lengths. Thus, for estimating F_θ, we need to properly average over the trajectory length. You should pass the tests for `compute_fisher_matrix`. The test solutions were computed using the default lambda value of 1e-3. We don't have tests for `sample` so make sure that is correct before moving on.

TODO Denote V^θ as the objective function $V^\theta = \mathbb{E}\left[\sum_{h=0}^{H-1} r(s_h, a_h) | s_0 \sim \mu_0, a_h \sim \pi_\theta(\cdot|s_h)\right]$. Again be mindful that each trajectory might have different lengths due to early termination. Let us implement the policy gradient, i.e.,

$$\widehat{\nabla} V^\theta = \frac{1}{N} \sum_{i=1}^{N} \left[\frac{1}{H} \sum_{h=0}^{H-1} \nabla_\theta \ln \pi_\theta(a_h^i|s_h^i) \left[\left(\sum_{t=h}^{H-1} r_t^i \right) - b \right] \right],$$

where b is a constant baseline $b = \sum_{i=1}^{N} R(\tau^i)/N$, i.e., the average total reward over a trajectory. Go to `utils.py` to implement this PG estimator in `compute_value_gradient`. There are tests in `test.py` for this function.

1.4 Implement the step size

With $\widehat{F}_\theta$ and $\widehat{\nabla}_\theta V^\theta$, recall that NPG has the following form:

$$\theta' := \theta + \eta \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta.$$

We need to specify the step size η here. Recall the trust region interpretation of NPG. We perform incremental update such that the KL divergence between the trajectory distributions of the two successive policies are not that big. Recall that the KL divergence $KL(\rho^{\pi_\theta}|\rho^{\pi_{\theta'}})$ can be approximate by Fisher information matrix as follows (ignoring constant factors):

$$KL(\rho^{\pi_\theta}|\rho^{\pi_{\theta'}}) \approx (\theta - \theta')^\top F_\theta(\theta - \theta').$$

As we explained in the lecture, instead of setting learning rate as the hyper-parameter, we set the trust region (which has a more transparent interpretation) size as a hyper parameter, i.e., we set δ such that:

$$KL(\rho^{\pi_\theta}|\rho^{\pi_{\theta'}}) \approx (\theta - \theta')^\top \widehat{F}_\theta(\theta - \theta') \leq \delta.$$

Since $\theta' - \theta = \eta \widehat{F}_\theta^{-1}\widehat{\nabla}V^\theta$, we have:

$$\eta^2(\widehat{\nabla}V^\theta)^\top \widehat{F}_\theta^{-1}\widehat{\nabla}V^\theta \leq \delta.$$

Solving for η we get $\eta \leq \sqrt{\frac{\delta}{(\widehat{\nabla}V^\theta)^\top \widehat{F}_\theta^{-1}\widehat{\nabla}V^\theta}}$. We will just set $\eta = \sqrt{\frac{\delta}{(\widehat{\nabla}V^\theta)^\top \widehat{F}_\theta^{-1}\widehat{\nabla}V^\theta}}$, i.e., be aggressive on setting learning rate while subject to the trust region constraint. To ensure numerical stability when the denominator is close to zero, we add $\epsilon = 1e-6$ to the denominator so the expression we will use is

$$\eta = \sqrt{\frac{\delta}{(\widehat{\nabla}V^\theta)^\top \widehat{F}_\theta^{-1}\widehat{\nabla}V^\theta + \epsilon}} \tag{1}$$

TODO Now go to `utils.py` and implement these step size computation in `compute_eta`. Check your implementation with the tests in `test.py`.

1.5 Putting Everything Together

Now we can start putting all pieces together. Go to `train.py` to implement the main framework in `train`. In each iteration of NPG, we do the following 4 steps:

- Collect samples by rolling out N trajectories with the current policy using the `sample` function.
- Compute the fisher matrix using the gradients from the previous steps. Use the default value of λ.
- Compute the step size for this NPG step.
- Update the model paramaters, θ, by taking an NPG step.

In addition to the above 4 steps, keep track of the average episode reward in each iteration of the algorithm. The output of `train` should be the final model parameters and a list containing the average episode rewards for each step of NPG.

For this section, run the above algorithm with the parameters $T = 20$, $\delta = 0.01$, $\lambda = 0.001$, and $N = 100$. Plot the performance curve of each π_{θ_t} for $t = 0$ to 19. You should be able to do this by simply running `train.py` from the terminal. The training output θ will be saved in folder `learned_policy/NPG/` and will be needed in the subsequent DAgger task.

Hint Your algorithm should achieve average reward of over 190 in about 15 steps if implemented correctly.

Section 2: DAgger

Note DAgger has not yet been covered in class so far, but it will be covered in a future lecture.

In this section, we will implement one classical imitation learning methods – DAgger – and get you started on implementing it in PyTorch. We will use the NPG policy that you trained in the previous sections as the expert policy π^*. We will be focusing on the CartPole environment as well.

2.1 Background and Setting

The goal in imitation learning is, given a set of N expert datapoints $\mathcal{D}^* = \{s_i^*, a_i^*\}_{i=1}^N$ from some unknown, black-box expert policy π^*, we want to learn a policy π that is as good as the expert.

DAgger is a method of interactive imitation learning that is similar to behavioral cloning, but we can query the expert on state-action pairs seen when rolling out our policy. In this way, when rolling out policy, we can essentially get an expert rollout $R^* = \{s_i, \pi^*(s_i)\}_{i=1}^H$, and we can "aggregate" this rollout into our dataset and learn from all our experience collected so far.

Partial observation of the learning agent. To demonstrate that imitation learning is powerful, we will constrain the learner's observation to a subset of the expert's, yet the learner is still capable of attaining comparable performance. Specifically, recall that the state of CartPole comprises four components: Cart Position, Cart Velocity, Pole Angle, and Pole Angular Velocity. Our goal is to demonstrate that, although the expert observes the full state, the policy learned by DAgger works well even when the learning agent is only able to observe a subset of the state.

2.2 Implementation

You will need to implement functions in `dagger.py`, which contains functions defining our DAgger learning agents and the training script. Specifically, please implement the following functions:

- `DAgger/sample_from_logits`: This method should take in action logits (of size (B, D_a)), and sample an action according to the distribution defined by said logits. Specifically, you will need to sample from the distribution $\Pr(a) = \exp(\iota_a) / (\sum_{a'} \exp(\iota_{a'}))$, where ι denotes the logits.
- `DAgger/rollout`: This method takes the environment and the number of steps to roll out in the environment as inputs, and should return a set of data points $(s, \pi^*(s))$ by rolling out in the environment. Specifically, this can be done in two steps: (1) roll out by taking actions **according to the current policy** (see `self.policy` definition in the class), and (2) get **the expert's action** for each state (see `self.expert_policy` definition in the class).
- `DAgger/learn`: This method should take in a batch of state and actions and perform a step of gradient descent over the batch. Specifically, you need to compute the action logits for the given states using the current policy, compute the cross entropy loss (see `self.loss`) using the logits and the expert actions, and perform one step of gradient descent.
- `experiment`: For a number of epochs, collect data while also training the agent on the new data. Specifically, first, we have to add our newly rolled out data into the dataset (see `dataset.py` for more information), and then create the dataloader to process our data. The dataloader has to be created every time before learning can begin due to the changing size of our dataset. Then, we perform supervised learning, i.e., we get batches from `dataloader` and perform gradient descent using `learner.learn`.

2.3 Testing

You can train the agent by running

```
python dagger.py --dagger_epochs 20 --num_rollout_steps 200
     --dagger_supervision_steps 20 --state_to_remove x
```

where `x` is an integer that could be 0,1,2, or 3, corresponding to which state component you want to remove (as we said earlier, we want to constrain the learner's observation).

Once done training, please test your agent via running `test_dagger.py`. You may also need to specify the argument `state_to_remove` for testing. The parameter `state_to_remove` should be consistent between the training and testing. In other words, we should always remove the same state component during both the training and testing. Other arguments for testing are the same ones specified by the `get_args` function in `utils.py`.

Please try to remove each of the four state components, and then report the average reward in the writeup. In other words, please run the training and testing for four times with `--state_to_remove` set to 0,1,2, and 3, respectively, and record the results. Please also save the four trained models with the names `x-CartPole-v0.pt` (where x=0,1,2, or 3 representing the removed state component) under the folder `learned_policies/dagger/`.

Please state in the writeup what you have discovered from these four trials. Which state component can be removed without compromising the performance, and which one impacts the performance the most when removed?

Hint You may want to verify your DAgger implementation by running the following to train a DAgger agent with full state information:

```
python dagger.py --dagger_epochs 20 --num_rollout_steps 200
               --dagger_supervision_steps 20
```

If your implementation is good enough, the agent will achieve an average reward of at least 190.

Section 3: Submission

```
YOUR_NET_ID/
├── learned_policies/
│   ├── dagger
│   │   ├── 0-CartPole-v0.pt
│   │   ├── 1-CartPole-v0.pt
│   │   ├── 2-CartPole-v0.pt
│   │   └── 3-CartPole-v0.pt
│   └── NPG
│       └── expert_theta.npy
├── answers.pdf
├── common.py
├── dagger.py
├── dataset.py
├── README.md
├── requirements.txt
├── test_dagger.py
├── test_info.pkl
├── test.py
├── train.py
└── utils.py
```

where `answers.pdf` should contain your plot for Section 1.5 and your discussion for Section 2.3.