Assignment 2: Policy Search

In this assignment you will implement two policy search algorithms, the cross-entropy method (CEM) [1], and the policy gradient method (PG) [2]. The template code is available at https://github.com/xbpeng/rl_assignments. Installation instructions are provided in `README.md`. All of the files that need to be changed in this assignment are located in the `a2/` directory. No files outside of this directory should be modified. Locations where code needs to be modified are labeled `TODO`.

Policy Search

Policy search encompasses a broad family of reinforcement learning algorithms where a policy π is trained to maximize the expected return by optimizing the following objective:

$$\arg\min_{\theta} \ \mathbb{E}_{\tau \sim p(\tau|\pi_\theta)} \left[\sum_{t=0}^{T-1} \gamma^t r_t \right], \tag{1}$$

where $\tau \sim p(\tau|\pi_\theta)$ denotes the trajectory distribution induced by executing a policy π_θ with parameters θ. The return is calculated as the sum of the discounted per-timestep rewards r_t, with a discount factor γ.

1 Cross-Entropy Method

One way to solve the policy search problem is by using black-box optimization techniques. The CEM algorithm [1], is a black-box optimization method, where Equation 1 is optimized by directly searching in the parameter space of the policy. The pseudo-code for CEM is available in Algorithm 1.

The algorithm searches for the optimal set of parameters by sampling candidate parameters from a search distribution $q^i(\theta)$. At every iteration i, CEM samples n candidate parameters $\{\theta_j\}$ from the current search distribution $q^i(\theta)$. The performance $J(\theta_j)$ of each set of parameters if evaluated by using those parameters for the policy to generate rollouts. The performance of the policy is then given by the mean return of the rollouts. Note, for CEM, no discounting needs to be applied when calculated the policy's expected return (i.e. $\gamma = 1$). Once the performance of every candidate has been determined, a set of m *elite* samples $\{\hat{\theta}_j\}$ are selected from the candidates, corresponding to the m candidates with the highest return. A new search distribution distribution $q^{i+1}(\theta)$ is constructed by maximizing the log-likelihood of the elite samples:

$$q^{i+1} = \arg\max_{q} \ \frac{1}{m} \sum_{j=1}^{m} \log q(\hat{\theta}_j). \tag{2}$$

ALGORITHM 1: CEM

$q^0 \leftarrow$ initialize search distribution

for iteration $i = 0, ..., k-1$ **do**
  Sample parameters $\theta_1, ..., \theta_n \sim q^i(\theta)$
  Evaluate performance of samples $J(\theta_1), ..., J(\theta_n)$
  Select elite samples with highest performance $\hat{\theta}_1, ..., \hat{\theta}_m$

  Update search distribution with elite samples:
  $q^{i+1} = \arg\max_q \ \frac{1}{m} \sum_{j=1}^{m} \log q(\hat{\theta}_j)$
end for

return best performing θ

In this assignments, the search distribution will be represented by a Gaussian $q = \mathcal{N}(\mu, \Sigma)$, with mean μ and a diagonal standard deviation matrix $\Sigma = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_d)$. To run CEM on the `cheetah` task, use the following command:

```
python run.py --mode train \
--env_config data/envs/dm_cheetah.yaml \
--agent_config a2/dm_cheetah_cem_agent.yaml \
--log_file output/log.txt \
--out_model_file output/model.pt \
--max_samples 200000000 \
--visualize
```

Once a model has been trained, you can load a checkpoint and test the model with the following command:

```
python run.py --mode test \
--env_config data/envs/dm_cheetah.yaml \
--agent_config a2/dm_cheetah_cem_agent.yaml \
--model_file output/model.pt \
--visualize
```

1.1 Candidate Samples

In `a2/cem_agent.py`, implement the `_sample_candidates(n)` method, which is used to sample `n` candidate parameters from the search distribution. The search distribution is a Gaussian with mean `self._param_mean` and standard deviation `self._param_std`. The output should be a set of candidates, represented by a tensor with dimensions $[n, d]$, where d is the size of the parameter vector.

1.2 Candidate Evaluation

In `a2/cem_agent.py`, implement the `_eval_candidates(candidates)` method, which is used to evaluate the performance of a given set of candidate parameters. The function `torch.nn.utils.vector_to_parameters` can be used to copy a vector of parameters to the model for the policy, which is represented by `self._model`. Once the parameters are set, the performance of the model can be evaluated using `self._rollout_test(num_eps)`, where `num_eps` specifies the number of episodes used for testing each candidate. The number of episodes that should be used per candidate is given by `self._eps_per_candidate`. Record the average return and average episode length of each candidate in the output variables `rets` and `ep_lens`.

1.3 Search Distribution

In `a2/cem_agent.py`, implement the `_compute_new_params(self, params, rets)` method, which constructs a new search distribution given a set of candidate parameters `params`, and the expected return `rets` of each candidate. This can be done by fitting a new Gaussian distribution that maximizes the log-likelihood of the *elite* samples. The number of elite samples to select from the candidates is determined by `self._elite_ratio`, which specifies the ratio of candidates to be selected as elite samples. The mean and standard deviation of the new search distribution should be stored in the output variables `new_mean` and `new_std`.

In practice, to prevent the standard deviation of the search distribution from becoming too small, it is often more effective to clamp the standard deviation to some minimal value. In your implementation, after a new distribution has been fitted, clamp the standard deviation of each parameter to be at least `self._min_param_std`.

1.4 Tasks

Train CEM policies for the `cheetah` task using the env config files `data/envs/dm_cheetah.yaml` and the corresponding agent config file `a2/dm_cheetah_cem_agent.yaml`. Train each model for at least 200 million timesteps, this can be specified in using the argument `--max_samples 200000000`. Tune the hyperparameters in the agent config file `a2/dm_cheetah_cem_agent.yaml`, so that the policies reach a return of at least 300. Plot a learning curve of the performance of the policy using plotting script `tools/plot_log/plot_log.py`.

2 Policy Gradient

The policy gradient algorithm is a class of policy search methods that aims to optimize Equation 1 through gradient ascent [2]. However, since the objective in Equation 1 is often non-differentiable, PG algorithms leverage an empirical gradient estimator using the score function. In this assignment, we will be using the *reward-to-go* formulation of the policy

gradient:

$$\nabla_\pi J(\pi) = \mathbb{E}_{\mathbf{s}\sim d_\pi(\mathbf{s})}\mathbb{E}_{\mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\mathbb{E}_{\tau\sim p(\tau|\pi,\mathbf{s}_0=\mathbf{s},\mathbf{a}_0=\mathbf{a})}\left[\nabla_\pi \log\pi(\mathbf{a}|\mathbf{s})\left(\sum_t \gamma^t r_t - V^\pi(\mathbf{s})\right)\right]. \quad (3)$$

The pseudo-code for the policy gradient method is available in Algorithm 2.

ALGORITHM 2: Reward-to-Go Policy Gradient

$\theta \leftarrow$ initialize policy parameters
$V \leftarrow$ initialize value function parameters

while not done **do**
  Sample trajectory τ from policy $\pi_\theta(\mathbf{a}|\mathbf{s})$
  Fit value function $V(\mathbf{s})$

  for every timestep t **do**
    $\nabla_t \leftarrow \left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} - V(\mathbf{s})\right) \nabla_\theta \log\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$
  end for

  $\nabla_\theta J(\pi_\theta) \approx \frac{1}{T}\sum_{t=0}^{T-1} \nabla_t$
  Update policy $\theta \leftarrow \theta + \alpha\nabla_\theta J(\pi_\theta)$
end while

return policy π_θ

To run PG on the `cheetah` task, use the following command:

```
python run.py --mode train \
--env_config data/envs/dm_cheetah.yaml \
--agent_config a2/dm_cheetah_pg_agent.yaml \
--log_file output/log.txt \
--out_model_file output/model.pt \
--max_samples 200000000 \
--visualize
```

Once a model has been trained, you can load a checkpoint and test the model with the following command:

```
python run.py --mode test \
--env_config data/envs/dm_cheetah.yaml \
--agent_config a2/dm_cheetah_pg_agent.yaml \
--model_file output/model.pt \
--visualize
```

2.1 Reward-to-Go

In `a2/pg_agent.py`, implement the `_calc_return(r, done)` method, which is used to calculate the return (i.e. reward-to-go) at every timestep. The input `r` is a tensor containing the per-timestep rewards, and `done` is a boolean tensor that indicates if a timestep is the last timestep of an episode. The output should be a tensor `return_t`, which contains the reward-to-go at every timestep $\sum_t \gamma^t r_t$. The discount factor is specified by `self._discount`.

Note, `r` can contain reward from multiple episodes concatenated together. For example, if we have three episodes, with rewards (r_0^0, r_1^0, r_2^0), (r_0^1, r_1^1), (r_0^2, r_1^2, r_2^2), then `r` will contain:

$$(r_0^0, r_1^0, r_2^0, r_0^1, r_1^1, r_0^2, r_1^2, r_2^2),$$

and `done` will contain:

(False, False, True, False, True, False, False, True)

When computing the return, only sum rewards from the same episode, do not sum the reward across multiple episodes together. Use `done` to determine the episode boundaries.

2.2 Advantage

In `a2/pg_agent.py`, implement the `_calc_adv(norm_obs, ret):` method, which is used to calculate the advantage at every timestep for updating the policy. The input `norm_obs` is the normalized observations at every timestep, and `ret` is the returns at every timestep. The output should be a tensor `adv` containing the advantage at every timestep. The advantage $A(\mathbf{s})$ at a given state $\mathbf{s}$ is defined as the difference between the return observed by the agent and the predicted return from the value function $V^\pi(\mathbf{s})$,

$$A(\mathbf{s}) = \left(\sum_t \gamma^t r_t\right) - V^\pi(\mathbf{s}), \tag{4}$$

where r_t denotes the rewards observed in all future timesteps starting at state $\mathbf{s}$.
The method `self._model.eval_critic(norm_obs)` can be used to evaluate the value function with a set of normalized observations `norm_obs`.

2.3 Critic Loss

In `a2/pg_agent.py`, implement the `_calc_critic_loss(norm_obs, tar_val)` method, which calculates the loss used to train the value function (i.e. critic). The input consists of the normalized observations `norm_obs`, and the target values `tar_val`, which are the returns observed by the agent at the corresponding states. The output should then be the loss for updating the value function. The loss is determined by the average least-squares error between the predictions from the value function $V(\mathbf{s}_i)$ and the target values $\mathbf{y}_i$,

$$l(V) = \mathbb{E}_{\mathbf{s}_i, y_i \sim \mathcal{D}} \left[||y_i - V(\mathbf{s}_i)||^2 \right], \tag{5}$$

where $\mathcal{D}$ is the dataset collected by the agent.

2.4 Actor Loss

In `a2/pg_agent.py`, implement the `_calc_actor_loss(norm_obs, norm_a, adv)` method, which calculates the loss used to train the policy (i.e. actor). The input consists of the normalized observations `norm_obs`, the normalized actions `norm_a`, and the advantages `adv` at each timestep. The output should then be the loss for updating the policy. The policy gradient loss is given by,

$$l(\pi) = -\mathbb{E}_{\mathbf{s}_i, \mathbf{a}_i, A_i \sim \mathcal{D}} \left[A_i \log \pi(\mathbf{a}_i | \mathbf{s}_i) \right]. \tag{6}$$

Note, since the objective is to *maximize* the policy's expected return, a negative sign is added to the loss so that minimizing $l(\pi)$ leads to maximizing the return. The method `self._model.eval_actor(norm_obs)` can be used to evaluate the policy with a set of normalized observations `norm_obs`, which will then return the action distribution of the policy $\pi(\mathbf{a}|\mathbf{s})$.

2.5 Tasks

Train PG a policies for the agent config file `a2/dm_cheetah_pg_agent.yaml`. Train each model for at least 200 million timesteps. Tune the hyperparameters in the agent config file `a2/dm_cheetah_pg_agent.yaml`, so that the policies reach a return of at least 300. Plot a learning curve of the performance of each using the plotting script `tools/plot_log/plot_log.py`.

3 Bonus

1 bonus point will be award to the submission that achieves the best performance on the `cheetah` tasks using the policy gradient method. To improve the performance of policy gradient, you can do more extensive tuning of the hyperparameters, as well as make any changes to the algorithm in `pg_agent.py`. Submit any modifications made to the PG algorithm in a separate file `bonus_pg_agent.py`, as well as a text file `bonus.txt` detailing the modifications you made.

Submission

Your submission should contain the following files:

- `cem_agent.py`: code changes.
- `dm_cheetah_cem_agent.yaml`: tuned hyperparameters for the `cheetah`.
- `cheetah_cem_model.pt`: trained model for the `cheetah` task.
- `cheetah_cem_log.txt`: training log for the `cheetah` task.
- `cheetah_cem_curve.png`: image of the learning curve for the `cheetah` task.

- `pg_agent.py` : code changes.
- `dm_cheetah_pg_agent.yaml` : tuned hyperparameters for the `cheetah` .
- `cheetah_pg_model.pt` : trained model for the `cheetah` task.
- `cheetah_pg_log.txt` : training log for the `cheetah` task.
- `cheetah_pg_curve.png` : image of the learning curve for the `cheetah` task.

- `bonus_pg_agent.py` : modified version of the PG agent for the bonus component.
- `bonus.txt` : a text file detailing any modifications you made in `bonus_pg_agent.py` for improving the performance of the algorithm.

All files should be stored in to a directory named `a1` , and then zip the directory for submission. Do not add any additional subdirectories.

References

[1] R. Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1:127–190, 1999.

[2] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.