Lab 4: Policy Optimization Algorithms

Deep Reinforcement Learning Bootcamp

August 26-27, Berkeley CA

1 Introduction

In this lab, you will implement various policy optimization algorithms.

2 Environment Setup

You should have your environment set up as specified in the pre-lab setup instructions.

3 Policy Gradient

If you are already familiar with policy gradient, you can jump to Section 3.2 directly.

3.1 Background

We will start with the standard policy gradient algorithm. This is a batch algorithm, which means that we will collect a large number of samples per iteration, and perform a single update to the policy using these samples.

Recall that the formula for policy gradient is given by

$$\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))\right] \tag{1}$$

where:

- π_θ is a stochastic policy parameterized by θ;
- γ is the discount factor;
- s_t, a_t, and r_t are the state, action, and reward at time t, respectively;
- T is the length of a single episode;
- $b(s_t)$ is a baseline function which only depends on the current state s_t (but does not depend on, say, the action a_t or future states $s_{t'}$ for $t' > t$).

- R_t is the discounted cumulative return, given by $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$. Strictly speaking, the unbiased gradient should use $\gamma^{t'}$ instead of $\gamma^{t'-t}$. However the gradient of the former term typically has a very weak signal from later time steps, and hence we typically use the latter term. The quantity $R_t - b(s_t)$ is an estimator of the *advantage* of taking action a_t in state s_t. This is also denoted as $\hat{A}(s_t, a_t)$.

Since this formula is an expectation, we cannot use it directly. It is more useful to consider a sampling-based estimator:

$$\nabla_\theta \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T} \gamma^t r_t \right] \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) \hat{A}(s_t^i, a_t^i) \tag{2}$$

where we assume that in each iteration, N trajectories are sampled: $\tau_1, \ldots, \tau_N$, and each trajectory τ_i is a list of states, actions, and rewards: $\tau_i = \{s_t^i, a_t^i, r_t^i\}_{t=0}^{T_i}$.

3.2 A Simple Program

We will start with a simple program, located at `simplepg/rollout.py`. This file lays out the general logic to simulate the interaction between a policy and an environment, and you can find traces of it in all later files. No coding is required for this step, and you only need to understand the overall structure of the code.

Start by running the following:

```
./docker_run.sh simplepg/rollout.py Point-v0
```

This will run a linear policy with randomly initialized parameters on a simple point-reaching environment. In this environment, a point will be spawned at a random location at the beginning of each episode and the goal is to move to center as soon as possible. The action space for this environment is a 2-dimensional continuous vector that denotes the desired amount of movement in x, y coordinates: (dx, dy). [1] Since the action space is continuous, we choose to use multivariate Gaussian distributions to be our action distribution.

A quick walkthrough of the code:

`point_get_action` samples a random continuous action given the current policy parameter `theta`.

The `main` function is the entry point of the entire program. It first checks which environment is selected, and set corresponding environment-specific variables. Then, it initializes `theta` to some random value, and run an infinite loop where we alternate between calling `get_action` to sample an action given the current state, and calling `env.step` to advance the state of the environment given the sampled action.

[1] dx and dy are both clipped to lie in $[-0.025, 0.025]$.

3.3 Computing $\nabla_\theta \log \pi_\theta(a|s)$ for Continuous Actions

Now we will switch to another file, `simplepg/main.py`, which provides a skeleton Policy Gradient implementation with a couple key pieces missing. In this file, we will fill in those components to make a working implementation.

Recall that the policy gradient estimator is $\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\hat{A}(s_t^i, a_t^i)$, which is an empirical mean of gradient terms $\nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\hat{A}(s_t^i, a_t^i)$. We will first implement a function that computes part of a gradient term given any s, a: $\nabla_\theta \log \pi_\theta(a|s)$.

The action a follows a multivariate Gaussian distribution with identity covariance matrix. The mean is given by $\mu = \theta^T \tilde{s}$, where $\tilde{s}$ is the vector obtained by appending 1 to the state s, so that we don't need a separate bias term. In `point_get_logp_action`, we provide an implementation of computing $\log \pi_\theta(a|s)$ for this policy parametrization.

Try running this file:

```
./docker_run.sh simplepg/main.py Point-v0 --use-baseline False
```

You will notice a message saying "Error: Gradient check didn't pass!" This is because the gradient has not been implemented. In this part, your task is to fix this error by filling in `point_get_grad_logp_action`, which should compute $\nabla_\theta \log \pi_\theta(a|s)$.

Try to work out what the gradient should be (the solution is in the footnote[2]).

After finishing this part, the error message should now be replaced by "Gradient check passed!". However, you should notice another message "Error: test for __main__.compute_update didn't pass!" This will be handled in the next part.

3.4 Accumulating Policy Gradient

This part will compute a single contribution to the overall policy gradient, given by $\nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))$. You will need to fill in the function `compute_update`. You should only need to make use of the input arguments given to this function. Note that R_t satisfies the recurrence relation: $R_t = \gamma R_{t+1} + r_t$ with initial condition $R_{T+1} = 0$.

After finishing this part, the error message should be replaced by "Test for __main__.compute_update passed!" You've now finished implementing a very basic policy gradient algorithm!

To visualize the policy while training, run the script with flag `--render True`:

```
./docker_run.sh simplepg/main.py Point-v0 --render True --use-baseline False
```

Right now, our algorithm learns rather slowly. It makes very little progress initially, and then gradually improves. It should reach an average return of about -25 after 100 iterations. We can improve the performance by using a baseline

[2] $\nabla_\theta \log \pi_\theta(a|s) = (a - \theta^T \tilde{s})\tilde{s}^T$. You can use `np.outer` to compute outer products.

(currently, the default baseline is constantly zero), which will be covered in the next part.

3.5 Time-Dependent Baseline

In this part, we will implement a simple time-dependent baseline, i.e. $b(s_t) = b_t$ for some learnable parameter b_t. We simply set it to the average of all R_t's among the trajectories collected from the previous iteration. If there are no trajectories long enough at a certain time step, we can set the baseline to 0. To finish this part, finish the implementation of `compute_baselines`.

After completing this part, run the following command:

```
./docker_run.sh simplepg/main.py Point-v0 --render True
```

Your algorithm should be able to achieve an average return of above -25 in around 30 iterations, and above -21 in around 60 iterations.

3.6 Computing $\nabla_\theta \log \pi_\theta(a|s)$ for Discrete Actions

In this part, we will switch gears and consider a cart-pole balancing task, which requires discrete actions. We will see that it's very easy to adapt our implementation to work with discrete action space.

To get a feeling of the cart-pole balancing environment, we can run the following command to see how does a random agent perform on this environment:

```
./docker_run.sh simplepg/rollout.py CartPole-v0
```

In this environment, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

The probability of an action a, +1 or -1 force, is given by $\text{softmax}(\theta^T \tilde{s})_a$, where $\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$.

Try running the following command:

```
./docker_run.sh simplepg/main.py CartPole-v0 --render True
```

You will notice that the gradient check failed again, since we need to implement the gradient for the linear cart-pole policy. Your task is to fill in `cartpole_get_grad_logp_action`. Again, try to work out what the gradient should be (the solution is given in the footnote[3]).

After correctly implementing the method, your code should be able to improve its performance (the desired average return should be above 180).

[3] $\nabla_\theta \log \pi_\theta(a|s) = (e_a - \pi_\theta(\cdot|s))\tilde{s}^T$, where e_a is a one-hot vector with all entries zero except in the ath entry, where the value is 1.

3.7 Hyperparameter tuning

The script `simplepg/main.py` supports configuring several hyperparameters using command line arguments. For example, the following command will run the algorithm on cart-pole with batch size 1000, discount factor 0.95, and learning rate 0.01:

```
./docker_run.sh simplepg/main.py CartPole-v0 \
    --batch_size 1000 \
    --discount 0.95 \
    --learning_rate 0.01
```

Play with different choices of the hyperparameters. Can you make it learn faster than the default options? Then, repeat the same experiment with the previous environment, `Point-v0`. Is there a combination of hyperparameters that works well for both environments?

3.8 Natural Gradient

In this part we will implement a basic version of natural policy gradient. Since the number of parameters is relatively small, we can explicitly compute the natural policy gradient (for larger scale policies, we can use conjugate gradient to avoid explicitly forming the entire matrix. This is covered in Section 5). Recall that this is given by

$$g_{\text{natural}} = F^{-1} g \tag{3}$$

where g is the policy gradient, and F is the Fisher information matrix, given by $F = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T\right]$. Note that this assumes θ is a flattened vector. Since we only have access to samples, we can estimate this using the collected samples. Implement this in `compute_fisher_matrix`. Note that $\mathbb{E}[xx^T] \neq \mathbb{E}[x]\mathbb{E}[x]^T$!

To test the implementation, you need to turn on natural gradient:

```
./docker_run.sh simplepg/main.py CartPole-v0 --natural True
```

Next, implement `compute_natural_gradient` which computes g_{natural} given F and g. To ensure that F is positive definite, use $F + \text{reg} \cdot I$ instead of F when computing the inverse. Note that the computation assumes that θ is a flattened vector, but the value returned from this method should have the same shape as `theta` (in our case, it is a matrix).

Finally, implement `compute_step_size`, which computes an adaptive step size. The step size can be chosen so that roughly, $D_{KL}(\pi_{\theta_{old}} \| \pi_\theta) \leq \epsilon$, where ϵ is a desired change in KL, specified by the hyperparameter `natural_step_size`. First, we use a second order approximation to the KL divergence, which gives $\frac{1}{2}\delta^T F \delta \leq \epsilon$, where δ is the update step on θ, given by $\delta = \alpha g_{\text{natural}}$. We'd like to

compute α (i.e. the return value of `compute_step_size`) so that the inequality is tight. Try to solve for the value of α. The solution is given in the footnote[4].

After completing these methods, you should observe much more stable performance improvement.

4 Advanced Policy Gradient

Hereafter, we will work with a more modular structure of the code, where different policies, action distributions, baselines, and algorithms only interact through well defined abstractions. For example, policies and baselines are implemented in `models.py`, distributions in `utils.py`, and algorithms in `pg.py`, `trpo.py`, and `a2c.py`. We also make use of Chainer for automatic, efficient gradient computation.

In this section, we will fill in a more structured implementation of policy gradient in `pg.py`.

4.1 Background

When using automatic differentiation libraries, it is convenient to rewrite Equation 2 as the gradient of a certain *surrogate loss* (we also flip the sign since typically we perform gradient descent rather than gradient ascent):

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\hat{A}(s_t^i,a_t^i) = \nabla_\theta \hat{L}_{\text{standard}}(\theta) \tag{4}$$

where

$$\hat{L}_{\text{standard}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\log \pi_\theta(a_t^i|s_t^i)\hat{A}(s_t^i,a_t^i) \tag{5}$$

We make use of a few additional improvements:

- To make the magnitude of the gradient more invariant, typically we also average over different time steps, in addition to averaging over different trajectories. To make it simpler, we typically maintain a flattened list of states, actions, and advantages indexed by $i = 1, 2, \ldots, M$, yielding

$$\hat{L}(\theta) = -\frac{1}{M}\sum_{i=1}^{M}\log \pi_\theta(a_i|s_i)\hat{A}(s_i,a_i) \tag{6}$$

- We further make use of Generalized Advantage Estimation [1] as covered in the lectures. This requires an additional hyperparameter λ.
- Rather than using plain gradient descent, we make use of momentum-based updates (specifically the Adam algorithm [2]).

[4] $\alpha = \sqrt{\frac{2\epsilon}{g_{\text{natural}}^T F g_{\text{natural}}}}$.

4.2 Implementing Policy Gradient

Open `pg.py`, which contains a partially implemented method `pg`. Your job is to finish this implementation.

The code starts by creating a pool of environment workers running in parallel:

```
with EnvPool(env_maker, n_envs=n_envs) as env_pool:
    ...
```

Then, within each iteration `iter`, it collects a list of trajectories, until `batch_size` samples are collected:

```
trajs = parallel_collect_samples(env_pool, policy, batch_size)
```

To compute the surrogate loss (so that we can later run backpropagation), we need to compute $\log \pi_\theta(a_t^i|s_t^i)$ and $\hat{A}(s_t^i, a_t^i)$ for each state-action pair. The following code computes the concatenated states and actions in all trajectories, as well as the estimated advantages:

```
all_obs, all_acts, all_advs, _ = compute_pg_vars(
    trajs, policy, baseline, discount, gae_lambda
)
```

Now, it is your turn to fill in the computation of `surr_loss` as in Equation 6, performed in the function `compute_surr_loss`. Since the advantages are already given in `all_advs`, the remaining task is to compute the log likelihood, and then form the surrogate loss. As a hint, take a look at what these methods do:

- `Distribution.logli` defined in `utils.py`.
- `F.mean` which is a shortcut for `chainer.functions.mean`.

A working implementation should take less than 5 lines of code.

4.3 Testing

First, make sure your implementation passes the inline tests. To see the algorithm in action, run the following script:

```
./docker_run.sh experiments/run_pg_cartpole.py
```

This will run the policy gradient algorithm on `CartPole-v0`, a classic control environment implemented in OpenAI Gym [3]. If your implementation is correct, you should observe steady improvement in the `AverageReturn` metric, and within 100 iterations you should be able to achieve an average return close to 200.

During any point of training, you can visualize your policy by running the following command in a separate terminal:

```
./docker_run.sh scripts/sim_policy.py data/local/pg-cartpole
```

You can also visualize the learning curve by running the following command:

```
./docker_run.sh viskit/frontend.py data/local/pg-cartpole
```

You should see a line similar to the following:

```
Done! View http://localhost:5000 in your browser
```

Follow the instruction and visit the address in your browser. You should see a window showing the learning progress so far like the following.

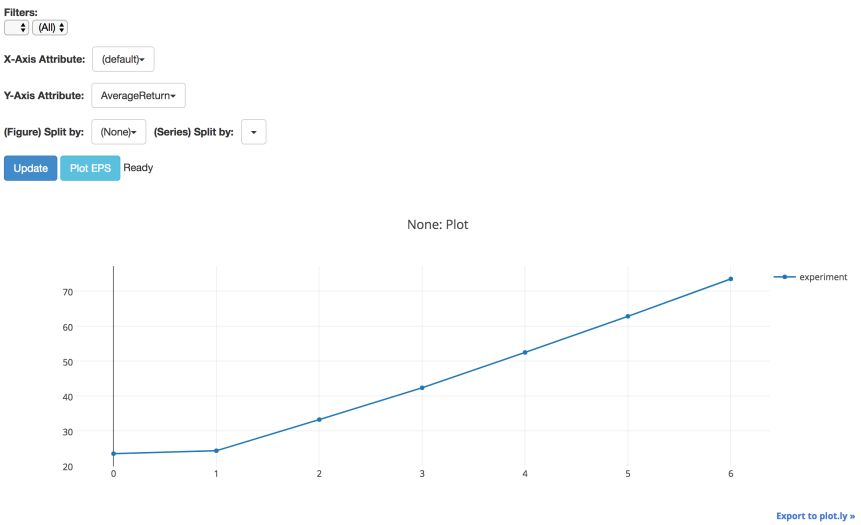

Figure 1: Visualizing learning curves.

You can toggle the "Y-Axis Attribute" to inspect other logged quantities. Explanation of the various logged quantities:

- `Iteration`: The current iteration number.
- `SurrLoss`: The current value of the surrogate loss.
- `Entropy`: The current average entropy of the policy computed on the collected samples. This value should decrease over time.
- `Perplexity`: The exponentiated entropy, roughly reflecting the number of actions the policy is making a decision in between.
- `AveragePolicyProb[i]`: The probability of taking the ith action, averaged over the collected samples.

- `AverageReturn`: The average total reward per episode over the past 100 trajectories (some of which may be from previous iterations). This is the main quantity to pay attention to, and should increase over time.
- `MinReturn`: The minimum total reward per episode over the past 100 trajectories.
- `MaxReturn`: The maximum total reward per episode over the past 100 trajectories.
- `StdReturn`: The standard deviation of the total reward per episode over the past 100 trajectories.
- `AverageEpisodeLength`: The average trajectory length over the past 100 trajectories (some of which may be from previous iterations).
- `MinEpisodeLength`: The minimum trajectory length over the past 100 trajectories.
- `MaxEpisodeLength`: The maximum trajectory length over the past 100 trajectories.
- `StdEpisodeLength`: The standard deviation of the trajectory length over the past 100 trajectories.
- `TotalNEpisodes`: The total number of episodes experienced so far throughout training. This and the next quantity are only updated when an entire trajectory has been collected, so there may be some delay in updating these numbers.
- `TotalNSamples`: The total number of time steps experienced so far throughout training.
- `ExplainedVariance`: The amount of variance in the returns explained by the baseline. Most of the time this quantity should be between 0 and 1 (although it can be negative, when the baseline is very off), and typically the larger this quantity is, the better. This is computed by $1 - \frac{\mathrm{Var}[R_t - b(s_t)]}{\mathrm{Var}[R_t]}$.

5 Trust Region Policy Optimization (TRPO)

If you are already familiar with TRPO, you can jump to Section 5.2 directly.

5.1 Background

Trust region policy optimization (TRPO) [4] improves upon policy gradient by explicitly controlling the change in the action distribution of the policy. Specifically it tries to solve the following constrained optimization problem:

$$\begin{aligned} \min_{\theta} \quad & L_{\theta_{\text{old}}}(\theta) \\ \text{s.t.} \quad & \mathbb{E}\left[D_{KL}(\pi_{\theta_{\text{old}}}(\cdot|s)\|\pi_{\theta}(\cdot|s))\right] \leq \epsilon \end{aligned} \tag{7}$$

where $L_{\theta_{\text{old}}}(\theta)$ is a surrogate loss function, defined as

$$L_{\theta_{\text{old}}}(\theta) = -\mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A(s_t, a_t)\right] \tag{8}$$

where we have the negative sign in the front since we are minimizing. The quantity ϵ in the optimization is a hyperparameter, controlling the maximum change in the average action distribution we allow per iteration.

To approximately solve the constrained optimization problem, we use a linear approximation to the surrogate loss, and a quadratic approximation to the constraint, yielding

$$\begin{aligned} \min_{\theta} \quad & g^T\theta \\ s.t. \quad & \frac{1}{2}\theta^T F(\theta_{\text{old}})\theta \leq \epsilon \end{aligned} \tag{9}$$

where $g = \nabla_\theta L_{\theta_{\text{old}}}(\theta)$, and $F(\theta_{\text{old}})$ is the Fisher information matrix of π_θ evaluated at θ_{old}.

The solution to this simplified problem is of the form $\theta^* = \lambda\tilde{\theta}$, where $\tilde{\theta} = F(\theta_{\text{old}})^{-1}g$ and λ is chosen so that the constraint is tight. This means that λ should satisfy

$$\frac{1}{2}(\lambda\tilde{\theta})^T F(\theta_{\text{old}})(\lambda\tilde{\theta}) = \epsilon \tag{10}$$

Rearranging this equation yields

$$\lambda = \sqrt{\frac{2\epsilon}{\tilde{\theta}^T F(\theta_{\text{old}})\tilde{\theta}}} \tag{11}$$

When θ is high dimensional, explicitly computing the Fisher information matrix $F(\theta_{\text{old}})$, let alone its inverse $F(\theta_{\text{old}})^{-1}$, can quickly become impractical. Fortunately, it turns out to be easy to compute the Fisher-vector product $F(\theta_{\text{old}})v$ for any given vector v. This enables us to use the Conjugate Gradient algorithm [5], which approximately solves a linear equation $x = A^{-1}b$ while only requiring access to an initial value x_0, target value b, and an operation $x \mapsto Ax$. After we obtain $\tilde{\theta}$, λ can be easily computed by another Fisher-vector product operation.

After we obtain θ^* for the simplified optimization problem, we perform a backtracking line search on the original problem to make sure that the exact KL constraint is satisfied, and that we have sufficient improvement in the original

objective. Since a line search procedure typically only takes a single objective but not constraints, we can incorporate the constraint into the objective via a barrier function:

$$L_{\text{barrier}}(\theta) = L_{\theta_{\text{old}}}(\theta) + 1e100 \cdot \max(D_{KL} - \epsilon, 0) \tag{12}$$

This way, we heavily penalize the loss function whenever the constraint is violated, but otherwise leave it the same.

5.2 Implementing TRPO

Open `trpo.py`, which contains a partially implemented method `trpo`. Your job is to finish this implementation. Look for `"*** YOUR CODE HERE ***"` to find the places you need to fill in.

Since TRPO is a rather complicated algorithm, and due to time constraint for the labs, we already provide most of the algorithm implementation. Your job is to finish the implementation of the inner helper function `f_loss_kl_impl`, which computes the surrogate loss and/or the average KL divergence, by filling in `compute_surr_loss` and `compute_kl`. The new distribution parameters (for $\pi_\theta(a_t|s_t)$) and the old distribution parameters (for $\pi_{\theta_{\text{old}}}(a_t|s_t)$) are already computed, and the flattened list of advantages is stored in the variable `all_advs`. These functions may be useful:

- `Distribution.likelihood_ratio`
- `Distribution.kl_div`

5.3 Testing

First, make sure your implementation passes the inline tests. We provide three scripts to further test your code, which require increasingly more time to train:

- The first one trains a discrete policy on `CartPole-v0`:

  ```
  ./docker_run.sh experiments/run_trpo_cartpole.py
  ```

  You should observe that the average return quickly converges to the optimal value 200. You can visualize the policy by running the following command in a separate terminal:

  ```
  ./docker_run.sh scripts/sim_policy.py data/local/trpo-cartpole
  ```

- The second one trains a continuous policy on `Pendulum-v0`, which is more challenging since it requires swinging up the pendulum and balancing it:

  ```
  ./docker_run.sh experiments/run_trpo_pendulum.py
  ```

Your implementation should be able to achieve an average return of at least around -180 after training for 100 iterations (your actual result may vary). You can visualize the policy by running the following command in a separate terminal:

```
./docker_run.sh scripts/sim_policy.py data/local/trpo-pendulum
```

- The last one trains a continuous policy for a planar cheetah-like robot. It uses a Roboschool environment `RoboschoolHalfCheetah-v1`, and requires you to install `roboschool` (this is already taken care for you if you are using the Docker setup):

  ```
  ./docker_run.sh experiments/run_trpo_half_cheetah.py
  ```

  This environment requires much more time to train. You should be able to achieve an average return of around 200 after 100 iterations, 1000 after 400 iterations, and 2000 after 2500 iterations (your actual result may vary).

  To visualize the policy, you need to set up a VNC Viewer as covered in the pre-lab setup instructions. You can visualize the policy by running the following command in a separate terminal (note that it's using `docker_run_vnc.sh` instead of `docker_run.sh`!):

  ```
  ./docker_run_vnc.sh scripts/sim_policy.py data/local/trpo-half-cheetah
  ```

  Remember that you need to connect to the displayed address from your VNC client.

  Since this experiment can be quite time consuming, if you ever need to interrupt the experiment and want to resume training later, you can use the following command:

  ```
  ./docker_run.sh scripts/resume_training.py data/local/trpo-half-cheetah
  ```

  Make sure not to call the `run_trpo_half_cheetah.py` script again, as it will wipe out the existing snapshots.

Explanation of the logged quantities, different from the ones displayed by previous algorithms:

- `ExpectedImprovement`: The expected improvement in the surrogate loss when updating the policy parameters, as given by the first-order approximation.
- `ActualImprovement`: The actual improvement in the surrogate loss.
- `ImprovementRatio`: The ratio between the actual improvement and the expected improvement. Ideally we want this value to be close to 1, which means that we have a good local approximation.

- `MeanKL`: The average KL divergence over the collected samples. This should be smaller (but not by much) than the specified step size.
- `AveragePolicyStd`: Only displayed for environments with continuous actions. The average standard deviation over all dimensions of the actions, across the collected samples.
- `AveragePolicyStd[i]`: The average standard deviation over the ith dimension of the actions, across the collected samples.

6 (Synchronous) Advantage Actor-Critic (A2C)

If you are already familiar with A2C, you can jump to Section 6.2 directly.

6.1 Background

The last algorithm we will implement is A2C, which is a synchronous implementation of Asynchronous Advantage Actor-Critic (A3C) [6]. Compared to the previous two algorithms, A2C is an online algorithm, which performs small updates to the policy using much smaller batches of data. This allows it to easily work with high-dimensional observations such as raw pixels [5].

Using small batches means that we will not have access to entire episodes during each update. Instead we use a value function $V(s)$ to estimate the sum of future rewards starting in state s. A detailed pseudocode of the algorithm is available in the original paper [6]. Our implementation mostly follows this pseudocode, except that there is no synchronization of parameters required, and we perform vectorized computation over a list of environments.

6.2 Implementing A2C

Open `a2c.py`, which contains a partially implemented method `a2c`. Your job is to finish this implementation. Look for `"*** YOUR CODE HERE ***"` to find all the places you need to fill in.

1. First, implement `compute_returns_advantages`, which takes in rewards r, termination flags d, estimated values V, estimated values of next states V_{next}, and discount factor γ, and outputs the computed empirical returns $\hat{R}$ and advantages $\hat{A}$. Each of the provided values are matrices, where the first index is time and the second index is the environment ID. Suppose that there are T time steps and N environments, and that the indices are 0-based. The output can be obtained by recursively computing from the

[5] This is also possible with, say, TRPO, but it introduces computation and memory challenges since TRPO requires a much larger batch size.

back:

$$\begin{aligned} R_T &\leftarrow V_{\text{next}} \\ R_t &\leftarrow r_t + (1 - d_t) \odot \gamma R_{t+1} \\ A_t &\leftarrow R_t - V_t \end{aligned} \tag{13}$$

where t ranges from $T-1$ down to 0, and $\odot$ denotes elementwise product (note that we do not include R_T in the returned values).

2. Next, we need to finish the implementation of the surrogate loss. This is similar to the surrogate loss used for standard policy gradient, except that we also have an entropy bonus with coefficient β (in the code this is `ent_coeff`), and an additional loss term for the value function with coefficient ν (in the code this is `vf_loss_coeff`). In total the surrogate loss should be

$$\hat{L}_{\text{total}}(\theta) = \hat{L}_{\text{pg}}(\theta) - \beta \cdot H(\pi_\theta) + \nu \cdot \hat{\mathbb{E}}\left[(R_t - V_\theta(s_t))^2\right] \tag{14}$$

Here, θ denotes the joint parameters of both the policy and the value function.

6.3 Testing

First, make sure your implementation passes the inline tests. We provide three scripts to further test your code. In all three cases, we will use A2C to train policies to play Atari games from raw pixel inputs.

- The first one trains a policy to play the game Pong, where we start from a partially trained policy:

  ```
  ./docker_run.sh experiments/run_a2c_pong_warm_start.py
  ```

  You should be able to see logs of average return after 2 to 3 epochs (each epoch consumes 10000 samples), since during the very first epoch none of the environments have finished one episode yet. If your implementation is correct, you should observe the average return rising from -21 to -13 in 10 epochs, 0 in 20 epochs, and around 20 in 30 epochs (your actual result may vary).

- The second one trains a policy to play the game Breakout from scratch:

  ```
  ./docker_run.sh experiments/run_a2c_breakout.py
  ```

  Although reaching convergence can take quite long (a few thousands of epochs in our experience), progress can be observed from very early on. With a proper implementation you should observe the average return improving from around 1 to around 10 after 50 epochs.

- The last one trains a policy to play Pong from scratch:

  ```
  ./docker_run.sh experiments/run_a2c_pong.py
  ```

  This can take a few hours before you see any signs of life. In our experience, the performance stayed at around -20 during the first 300 up to even 700 epochs. Then, it will gradually improve to around 20 in a few hundred more epochs. Be patient :)

Explanation of the logged quantities that are in addition to those logged by previous algorithms:

- `|VfPred|`: The average of the absolute value of the predicted values.
- `|VfTarget|`: The average of the absolute value of the target values.
- `VfLoss`: The value of the value function loss, without the scaling coefficient.

References

[1] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.

[2] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 1889–1897, 2015.

[5] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[6] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 1928–1937, 2016.