Reinforcement Learning Lab

Lesson 12: SAC

Alberto Castellini and Gabriele Roncolato

University of Verona
email: alberto.castellini@univr.it, gabriele.roncolato@univr.it

Academic Year 2024-25

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Environment Setup

Follow the instructions at https://github.com/Isla-lab/RL-lab/blob/main/README.md to setup the new miniconda environment.

- Set-up a new and separate conda environment for Spinning Up:

```
conda create -n spinningup python=3.6
conda activate spinningup
sudo apt-get update && sudo apt-get install libopenmpi-dev
```

- install the Spinning Up dependencies:

```
navigate to RL-lab/spinningup/spinningup
pip install opencv-python==4.1.2.30
pip install -e .
```

- Update the repository of the lab:

```
cd RL-Lab
git stash
git pull
git stash pop
```

Training the RL Agent

Follow the instructions at https://github.com/Isla-lab/RL-lab/blob/main/README.md to train and test the RL Agents.

- Remember to activate your miniconda environment:

```
conda activate spinningup
```

- To train a RL agent, run the *train.py* script located inside the *spinningup* folder using the following arguments:
  - *env*: the environment to train the RL agent on (required)
  - *algo*: the RL algorithm to be used during training (required)
  - *exp_name*: the name of the experiment, necessary to save the results and the agent weights (required)
  - *hid*: a list representing the neural network hidden sizes (default is [32, 32])
  - *epochs*: the number of training epochs (default is 50)
- Example: train Vanilla Policy Gradient (VPG), i.e., REINFORCE with Baseline, over the CartPole environment (once the training is complete, a performance graph is visualized):

```
python train.py --env CartPole-v1 --algo vpg --exp_name first_experiment
```

Spinning up: code flow

The previous command line executes the Python file *spinningup/train.py* which cascades the following Python files:

- *spinningup/spinup/run.py*: makes some parsing and executes the required experiments by running the algorithms in the following directories;
- *spinningup/spinup/algos/pytorch* or *spinningup/spinup/algos/tf1*: contain the algorithms (e.g., *vpg.py* and *sac.py*)
- *spinningup/spinup/algos/pytorch/sac/sac.py*: performs policy learning. To update policy and value functions it calls the update rule defined in related file contained in the directory *algorithms/*
- *spinningup/spinup/algos/pytorch/sac/core.py*: contains the classes for the actor and the critic nets
- *algorithms/sac.py*: contains the update rule of the ppo algorithm. This file is a template that must be completed.

Today Assignment

In today's lesson, we will implement the update rule of the SAC algorithm within the *spinningup* framework and we will run it on some gym environments. On which environments can the SAC algorithm be used? Check the tables in the next slides.
The file to complete is:

```
RL-lab/spinningup/algorithms/sac.py
```

Inside the file, three Python functions are partially implemented. The objective of this lesson is to complete them.

- **def loss_pi_fn(data, ac)**
- **def loss_q_fn(data, ac, ac_targ, gamma)**
- **def update_rule(data, q_optimizer, pi_optimizer, logger, ac, ac_targ, gamma, polyak)**

Your task is to fill in the code where there are #TODO, or "pass" words. Expected results can be found in:

```
RL-Lab/results/lesson_12_results.*
```

SAC to implement (https://spinningup.openai.com/en/latest/algorithms/sac.html)

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

SAC theory

Soft Actor Critic (SAC) is an **off-policy** algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches. A central feature of SAC is **entropy regularization**. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy.

- The entropy H of a stochastic variable x can computed from its distribution P according to $H(P) = E_{x\sim P}[-\log P(x)]$.
- In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes the RL problem to:

$$\pi^* = \arg\max_{\pi} \underset{\tau\sim\pi}{\mathrm{E}} \left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big) \right], \tag{1}$$

where $\alpha > 0$ is the trade-off coefficient.

SAC theory

Value and Q-functions are also changed to include the entropy bonuses

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \Big) \middle| s_0 = s \right], \tag{2}$$

$$Q^{\pi}(s, a) = \mathop{\mathrm{E}}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H\left(\pi(\cdot|s_t)\right) \middle| s_0 = s, a_0 = a \right], \tag{3}$$

V^{π} and Q^{π} are connected by:

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{a \sim \pi} \left[Q^{\pi}(s, a) + \alpha H\left(\pi(\cdot|s)\right) \right] \tag{4}$$

SAC theory

SAC concurrently learns a policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2} (no V function is used). Spinning Up implements a version with a fixed entropy regularization coefficient α, but the entropy-constrained variant is generally preferred by practitioners.

- Both Q-functions are learned with Mean Squared Bellman Error (MSBE) minimization, by regressing to a single shared target;
- The shared target is computed using target Q-networks, and the target Q-networks are obtained by polyak averaging the Q-network parameters over the course of training;
- The shared target makes use of the clipped double-Q trick (introduced in TD3).
- Full math details at https://spinningup.openai.com/en/latest/algorithms/sac.html

SAC theory: Q-function loss

The loss functions for the Q-networks in SAC are:

$$L(\phi_i, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_{\phi_i}(s,a) - y(r,s',d) \right)^2 \right], \tag{5}$$

where the target is given by

$$y(r,s',d) = r + \gamma(1-d) \left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s',\tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s'). \tag{6}$$

SAC theory: policy loss

The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize $V^{\pi}(s)$, which can be expanded out into

$$V^{\pi}(s) = \underset{a\sim\pi}{\mathrm{E}} \left[Q^{\pi}(s,a) + \alpha H\left(\pi(\cdot|s)\right) \right] \tag{7}$$

Which is

$$V^{\pi}(s) = \underset{a\sim\pi}{\mathrm{E}} \left[Q^{\pi}(s,a) - \alpha \log \pi(a|s) \right]. \tag{8}$$

The way the policy is optimized makes use of the **reparameterization trick**, in which a sample from $\pi_{\theta}(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise.
The policy is then optimized according to

$$\max_{\theta} \underset{s\sim\mathcal{D},\xi\sim\mathcal{N}}{\mathrm{E}} \left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_{\theta}(s,\xi)) - \alpha \log \pi_{\theta}(\tilde{a}_{\theta}(s,\xi)|s) \right] \tag{9}$$

SAC: implementation

1. First, you should read the code in files *spinningup/spinup/algos/pytorch/sac/sac.py* and *spinningup/spinup/algos/pytorch/sac/core.py* to understand the algorithm organization. These files must not be modified.
2. Every *update_every* steps (i.e., not every epoch) the *update* function is called, which, in turn, calls the *update_rule* function of in *algorithms/sac.py*.

```
def update_rule(data, q_optimizer, pi_optimizer, q_params, logger, ac, ac_targ,...)
```

3. The *update_rule* function first computes the loss function of the q-function, its gradient and updates its parameters (only one time)

```
loss_q, q_info = loss_q_fn(data, ac, ac_targ, gamma, alpha)
loss_q.backward()
q_optimizer.step()
```

then it does the same with the policy

```
loss_pi, pi_info = loss_pi_fn(data, ac, alpha)
loss_pi.backward()
pi_optimizer.step()
```

SAC: implementation, q-function loss

4. The q-function loss *loss_q* is computed by the function *loss_q_fn* which uses the information in the reply buffer *data*, the networks in the actor-critic object *ac* and the target networks in the actor-critic object *ac_targ*

```
def loss_q_fn(data, ac, ac_targ, gamma):
  o, a, r, o2, d = data['obs'], data['act'], data['rew'], data['obs2'], data['done'
  ...
```

5. You should first compute $Q_{\phi_{\text{targ},j}}(s', \tilde{a}')$ for $j = 1$ and $j = 2$ (see Eq. 6) and put them in variables *q1_pi_targ* and *q2_pi_targ*.
6. Then, you should compute the Bellman backup $r + \gamma(1-d)\left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right)$ (see Eq. 6) and put it in variable *backup*.
7. Finally, you should compute the MSE loss against Bellman backup (overall Eq. 5) for the two q-functions, put them in variables *loss_q1* and *loss_q2*, and put their sum in variable *loss_q*. This is the q-function loss to be returned.

SAC: implementation - policy loss

8. The policy loss *loss_pi* is computed by the function *loss_pi_fn* which uses the information in the reply buffer *data* and the networks in the actor-critic object *ac*

```
def loss_pi_fn(data, ac):
  o = data['obs']
  ...
```

9. According to Eq. 9 you should first compute the min between $Q_{\phi_1}(s, \tilde{a}_\theta(s, \xi))$ and $Q_{\phi_2}(s, \tilde{a}_\theta(s, \xi))$ and put it in *q_pi*
10. Then you should compute the difference between *q_pi* and $\alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi)|s)$, average over samples in the reply buffer, and compute the opposite (because Eq. 9 is a maximization). This value should be put in variable *loss_pi* and returned.

Gymnasium environments

Environment	**States**	**Actions**	Transition
MountainCar-v0	2 Continuous	3 Discrete	Deterministic
CartPole-v1	4 Continuous	2 Discrete	Deterministic
Acrobot-v1	6 Continuous	3 Discrete	Deterministic
LunarLander-v2	8 Continuous	4 Discrete	Deterministic
MountainCarContinuous-v0	2 Continuous	1 Continuous	Deterministic
Pendulum-v0	3 Continuous	1 Continuous	Deterministic
BipedalWalker-v3	24 Continuous	4 Continuous	Deterministic
FrozenLake-v0	16 Discrete	4 Discrete	Stochastic

How can we make SAC work on discrete state spaces? Can it work on discrete action spaces?

RL Algorithm capabilities

Algorithm	States	Actions	Transition	On/Off-policy	V/Q	π
Policy gradient	D	D	Needed	Planning	V	No
MC Control	D	D	No	On-policy	Q	No
Sarsa	D	D	No	On-policy	Q	No
Q-Learning	D	D	No	Off-policy	Q	No
Dyna-Q	D	D	Learned	Off-policy	Q	No
DQN	D/C	D	No	Off-policy	Q(2)	No
REINFORCE	D/C	D/C	No	On-policy	No	Yes
REINFORCE+B	D/C	D/C	No	On-policy	V	Yes
A2C	D/C	D/C	No	On-policy	V	Yes
MCTS	D	D	Needed	Planning	Q	No
PPO	D/C	C/D	No	On-policy	V	Yes
DDPG	D/C	C	No	Off-policy	Q	Yes
SAC	D/C	C/D	No	Off-policy	Q	Yes

Expected Results - MountainCarContinuous-v0 with SAC

Once the algorithm is complete it can be run with the command

```
python train.py --env MountainCarContinuous-v0 --algo sac --exp_name first_exp_sac
```

It produces textual output in the terminal and a chart with performance over training episodes

Epoch	50
AverageEpRet	-31.3
StdEpRet	0.436
MaxEpRet	-30.6
MinEpRet	-31.8
AverageTestEpRet	-0.187
StdTestEpRet	0.0229
MaxTestEpRet	-0.157
MinTestEpRet	-0.226
EpLen	999
TestEpLen	999
TotalEnvInteracts	2e+05
AverageQ1Vals	42.1
StdQ1Vals	1.19
MaxQ1Vals	43.6
MinQ1Vals	14.8
AverageQ2Vals	42.1
StdQ2Vals	1.19
MaxQ2Vals	43.6
MinQ2Vals	15
AverageLogPi	-0.645
StdLogPi	0.274
MaxLogPi	2.62
MinLogPi	-7.15
LossPi	-42.2
LossQ	3.18
Time	1.5e+03

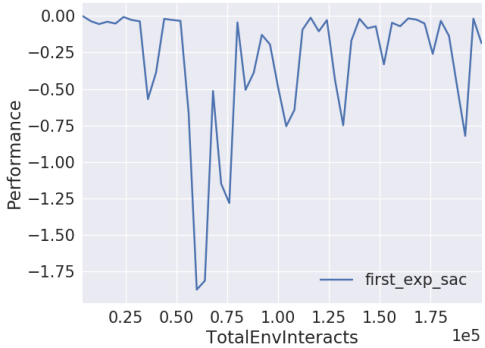

Figure: Mean return of SAC for Mountain Car.

Expected Results - MountainCarContinuous-v0 with SAC

When the experiment is concluded, open the related folder (*spinningup/spinningup/data/first_experiment_sac/* in our case) you will find the files:

- *config.json*: containing the configuration parameters
- *progress.txt*: containing the learning progress
- *model.pt*: a PyTorch checkpoint file that contains the trained policy network (actor) and possibly the value function (critic) for future reuse (see code below to load it)

```
from spinup.utils.test_policy import load_policy
env_fn = lambda: gym.make("YourEnv-v0")
pi, get_action = load_policy(fpath='path_to_saved_model', itr='last', deterministic
```

- *vars.pkl*: a dictionary containing info about the experiment (see code below to inspect it)

```
import pickle
with open('path_to/vars.pkl', 'rb') as f:
config = pickle.load(f)
print(config.keys())
```

Expected Results - MountainCarContinuous-v0 with SAC: policy testing

Finally, you can test the learned policy by running the file *spinningup/test.py*.

```
python test.py --exp_name first_exp_sac
```

Expected Results - MountainCarContinuous-v0 with PPO

A comparison with PPO can be performed

```
python train.py --env MountainCarContinuous-v0 --algo ppo --exp_name first_exp_mc_ppo
```

Epoch	49
AverageEpRet	92
StdEpRet	1.19
MaxEpRet	94.2
MinEpRet	90.3
EpLen	286
AverageVVals	81.7
StdVVals	0.000587
MaxVVals	81.7
MinVVals	81.7
TotalEnvInteracts	2e+05
LossPi	3.58e-09
LossV	84.2
DeltaLossPi	-0.00109
DeltaLossV	-0.0809
Entropy	0.322
KL	0.000859
ClipFrac	0.0065
StopIter	79
Time	98.9

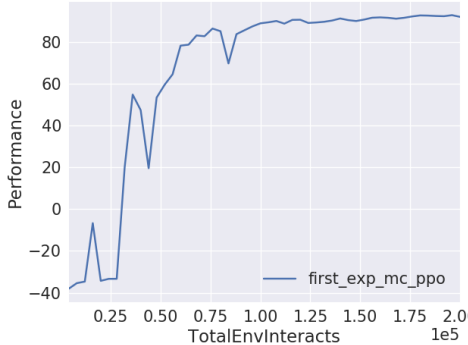

Figure: Mean return of PPO for Mountain car.

Training on other environments

Compare the performance of SAC, DDPG, PPO and VPG on all the environments in the table above for which the algorithms can be compared. E.g., how does it perform on Pendulum-v0?

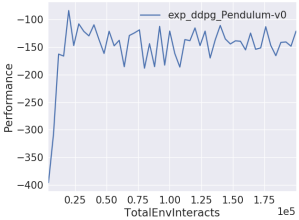

Figure: Mean return for DDPG on Pendulum-v0.

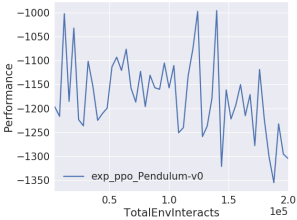

Figure: Mean return for PPO on Pendulum-v0.

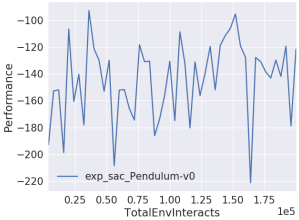

Figure: Mean return for SAC on Pendulum-v0.

Try to change the parameters of the algorithm and analyze the effect on the performance.

Interesting questions

- Does the code work with discrete state spaces? What can you change to make it work?
- Can you run the algorithms also on other gym environments? How?