Reinforcement Learning Project

Implementing SAC in a HumanoidStandup environment

Introduction

The purpose of this work is to implement the Soft Actor-Critic algorithm and to train and test it in the OpenAI Gym environment "HumanoidStandup-v2".

The used frameworks are Python and Tensorflow.

About SAC

In the following chapter, a short description of the main features of the SAC algorithm will be provided.

Maximum Entropy Reinforcement Learning

The SAC algorithm, introduced and described in detail in [1] and [2], is an off-policy reinforcement learning algorithm based on the maximum entropy framework, which means that the agent maximizes a target function which combines the expected return, as in traditional Reinforcement Learning, and the entropy of the policy; in other words, the agent tries to perform the task well while acting with a sizable degree of randomness.

This new target function provides the advantage of incentivizing the agent to explore more and explore around the higher-reward parts of the domain, while abandoning the lower-reward ones. Exploration is an important component of any RL process because it allows the agent to see a wider region of the domain and discover the most promising avenues to obtain a high reward: therefore by encouraging exploration the the learning process as a whole is improved.

The mathematical definition of the aforementioned target function is:

$$J(\pi)=\sum_{t=0}^{T} E_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)+\alpha H(\pi(\cdot|s_t))] \qquad (1)$$

Where r is the reward function, H is the entropy of the policy, and alpha is a hyperparameter called "temperature" which weighs the two components of the target function.

The traditional RL target function can be recovered from the maximum entropy one with $\alpha \rightarrow 0$. A discount factor gamma can be introduced in a similar way to traditional RL.

Soft Policy Iteration

SAC is derived from a variant of the policy iterarion algorithm applied to the maximum entropy framework, called "soft policy iteration". This algorithm iteratively alternates a policy evaluation step and a policy improvement step.

The **policy evaluation** step improves the estimate of the soft Q-function of the policy by using a modified version of the Bellman operator:

$$Q \leftarrow r(s_t, a_t) + \gamma E_{s_{t+1} \sim p}[V(s_{t+1})] \quad (2)$$

where

$$V(s_t) = E_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t | s_t)] \quad (3)$$

The **policy improvement** step updates the policy based on the current Q-function estimate. The policy is restricted to some set PI to ensure it is tractable, therefore the new policy resulting from the improvement is projected on PI according to the KL divergence.

Therefore the mathematical formulation of this step is:

$$\pi \leftarrow argmin_{\pi' \in PI} D_{KL}\left(\pi'(\cdot | s_t) \| \frac{\exp\left(\frac{1}{\alpha} Q^{\pi}(s_t, \cdot)\right)}{Z^{\pi}(s_t)}\right) \quad (4)$$

where Z is a normalization term that can be ignored as it does not contribute to the gradient.

While this algorithm is theoretically sound, it is impractical in a continuous domain because alternating the two steps while using function approximators such as neural networks for the policy and the Q-function is too expensive computationally.

Soft Actor-Critic

SAC is derived as an approximation of soft policy iteration which uses neural networks for the Q-function and the policy and optimizes the two networks alternatively with gradient descent rather than executing the soft policy iteration steps.

The **soft Q-function** is approximated by a neural network with parameter vector θ which is trained to minimize the soft Bellman residual:

$$J_Q(\theta) = E_{(s_t, a_t) \sim D}\left[\frac{1}{2}(Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma E_{s_{t+1} \sim p}[V_{\hat{\theta}}(s_{t+1})]))^2\right] = \ldots$$
$$\ldots = E_{(s_t, a_t) \sim D}\left[\frac{1}{2}(Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma E_{s_{t+1} \sim p}[E_{a_t \sim \pi}[Q_{\hat{\theta}}(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1} | s_{t+1})]]])\right. \quad (5)$$

which is derived from equations (2) and (3).

In addition to the Q-function estimator with parameters θ , this update rule uses another such estimator with parameters $\hat{\theta}$. This latter parameter vector is obtained as an exponential moving average of the values taken by θ over the course of the training process.

The **policy** is approximated by a different neural network with parameter vector ϕ which is trained to minimize the expected KL divergence:

$$J_\pi(\phi) = E_{s_t \sim D}[E_{a_t \sim \pi_\phi}[\alpha \log(\pi_\phi(a_t | s_t)) - Q_\theta(s_t, a_t)]] \quad (6)$$

Which is derived from equation (4).

In order to minimize (6), it is useful to express a_t , the output of the policy, in terms of a stochastic transformation dependent on the parameters ϕ via the *reparametrization trick*:

$$a_t = f_\phi(\epsilon_t; s_t) \quad (7)$$

where ϵ is a noise vector sampled from a Gaussian distribution.

Equation (6) is then rewritten as:

$$J_{\pi}(\phi)=E_{s_t \sim D}[E_{\epsilon_t \sim N}[\alpha \log(\pi_{\phi}(f_{\phi}(\epsilon_t;s_t)|s_t))-Q_{\theta}(s_t,f_{\phi}(\epsilon_t;s_t))]] \qquad (8)$$

which is then used as the loss function for the policy.

It is also possible to use multiple Q-function estimators, train each of them independently, and replace the Q_{θ} in equation (8) and the $Q_{\bar{\theta}}$ in equation (5) with the minimum of those estimations: this is known to reduce positive bias and speed up training significantly.

Implementation Details

In this chapter, the practical implementation of SAC developed for this project will be introduced and discussed in detail.

Function approximators

In this implementation it was chosen to use two Q-function estimators, as discussed a few paragraphs above.
Therefore, a total of five function approximators are created and trained: two Q-functions, the respective "target" Q-functions (corresponding to $Q_{\hat{\theta}}$), and the stochastic policy.

The Q-functions and the respective "target" estimators each consist entirely of one dense neural network that takes observation-action pairs as inputs and outputs the corresponding Q-values, while the policy samples actions from a multivariate Gaussian distribution whose means and standard deviations are provided by a dense neural network which outputs those parameters given observations as inputs. The policy is also capable of returning the logarithm of the probability of selecting each of the sampled actions, which is useful when training the policy, as shown by equation (6).

Additionally, a deterministic copy of the policy can be created at any time for the purposes of testing performance. Such deterministic copy directly outputs the means given by the neural network, without doing any random sampling.

Environment interaction step

The implementation alternates between environment interaction steps to accumulate experience and stochastic gradient descent steps to train the function approximators.

Interaction with the environment is carried out through the standard OpenAI Gym interface: given an action to perform, a single simulation step is executed and the reward signal and the new observation of the agent are returned.

The action selection criterion changes over the course of training: for a short period of time at the beginning of the process the next action is selected in an entirely random way by uniformly sampling the action space, then after this initial phase the next action is always chosen according to the stochastic policy. Having a completely random initial phase provides the benefit of focusing on pure exploration for the first few episodes, therefore collecting as much data as possible about the region around the initial state before the policy steers the search towards the areas it considers more promising.

The results of each interaction step are saved in a replay buffer, which stores the accumulated experience of the agent in the form of <observation, action taken, received reward, resulting new observation, terminal signal> records. The stored records are then sampled during the training steps. If the buffer's capacity limit is reached, the older records are overwritten by the new ones.

Training step

During one training step, each of the function approximators is trained given a batch of samples from the replay buffer as data.

Specifically:

- The policy is trained by applying the Adam optimizer on the following loss function, derived from equations (6) and (8):

  $$policyLoss = \alpha \log \pi(act|obs) - min_{i \in \{1,2\}} Q_i(obs, act) \tag{9}$$

  where obs is the observation stored in a replay buffer record and act and $\log \pi(act|obs)$ are the action and the respective log-probability sampled by the stochastic policy given obs.

- Each Q-function is trained independently by applying the Adam optimizer on the following loss function, derived from equation (5):

  $$\begin{gathered} qLoss = \frac{1}{2}(Q(obs, act) - target(nextObs, rew, done))^2 \\ \text{where} \\ target = rew + \gamma \cdot (1 - done) \cdot V(nextObs) \\ \text{and} \\ V(nextObs) = min_{i \in \{1,2\}} Qtarg_i(nextObs, nextAct) - \alpha \log \pi(nextAct|nextObs) \end{gathered} \tag{10}$$

  where $obs, act, nextObs, rew, done$ are the data contained in a replay buffer record (consider $done \in \{0,1\}$) and $nextAct$ and $\log \pi(nextAct|nextObs)$ are the action and the respective log-probability sampled by the stochastic policy given $nextObs$.

  The qLoss is identical in form to a standard MSE loss, and therefore it was implemented as such.

  Also note that $target$ is independent of which Q-function is being trained, therefore in the implementation it was computed once and reused for the training step of both Q-functions.

- Each "target" Q-function estimator is improved independently by updating its exponential average with the new value of the parameters of the corresponding Q-function. Therefore, by calling $\hat{\theta}$ the parameters of a "target" Q-function estimator and θ the parameters of the corresponding Q-function after a training step, the update rule is:

  $$\hat{\theta} \leftarrow (\tau\theta - (1-\tau)\hat{\theta}) \tag{11}$$

  where $0 \leq \tau \leq 1$ is a hyperparameter determining the weight of the new sample with respect to the others in the average.

Additional design choices were made regarding training:

First, environment interaction steps and training steps were alternated in batches of 50 steps each rather that as individual steps, in order to batch training samples together and let each version of the policy act for a period of time rather than a single step. Even though the steps are not alternated individually, this "batching" alternation still guarantees that the ratio between interaction and training steps is 1:1.

Second, the initial training steps were skipped for a short period of time in order to let the replay buffer fill to an acceptable size before attempting to sample from it. The no-training phase always ends before the random-action phase, so that the policy can get some initial training and move away from its random initialization before it is used to guide environment interaction.

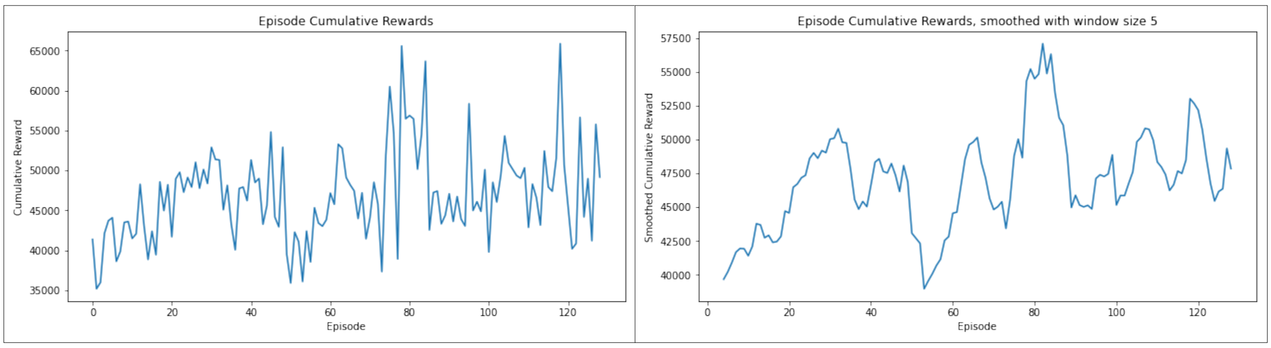
Tests

As mentioned in the introduction, the SAC implementation described above was trained and tested on the HumanoidStandup-v2 OpenAI Gym environment. It can be considered a complex environment relative to others in the Gym framework because of the comparatively higher dimensionality of the observation and action spaces and its infinite-reward quality. Furthermore, because it has no termination conditions, episodes always run until a time limit externally enforced by this implementation.

The training lasted 129 epochs[1], each of which was composed of one training episode during which the whole SAC algorithm described above was executed, and one test episode during which training was put on hold and the agent used the current policy deterministically.

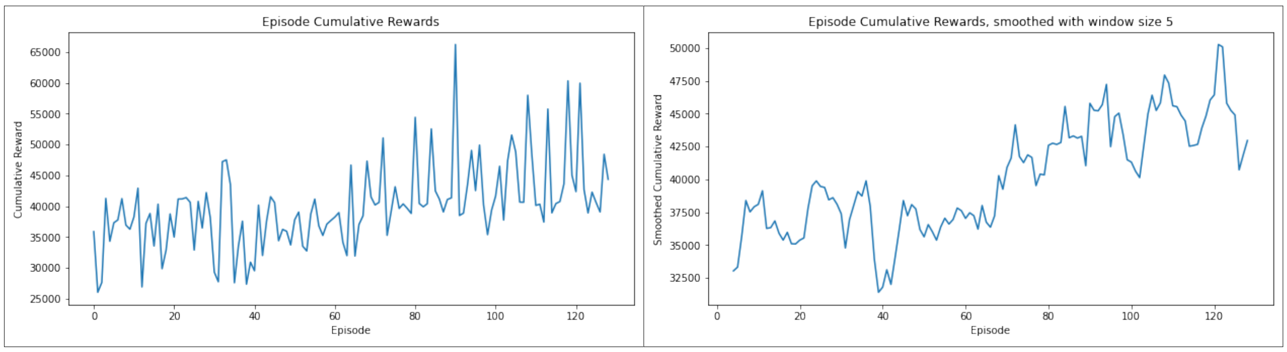
The only metric considered in this evaluation is the cumulative reward of each epoch for training episodes and testing episodes. Because this metric was observed to be quite noisy in this environment, a smoothed rolling-average version of the data will be provided in addition to the raw one.

The following are the plots of the cumulative reward of each training episode, in both raw and smoothed form:



1 The training was intended to last 175 epochs, but the process was prematurely killed by the operating system, most likely because of an Out Of Memory error. The data collected before being killed are still fully valid.

And the following are those of the test episodes, in raw and smoothed form:



It can be observed that the reward shows an increasing trend over time, albeit slow and irregular, showing that the agent is indeed capable of improving its performance. This trend is especially visible in the test episodes.

It is also noticeable that the reward of the training episodes tends to be higher and higher-variance compared to the test episodes. This is not surprising, because the training episodes use a stochastic policy instead of a deterministic one like the test episodes.

References

1: Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine, Soft Actor-Critic:Off-Policy Maximum Entropy Deep ReinforcementLearning with a Stochastic Actor, 2018
2: Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha ,Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, Sergey Levine, Soft Actor-Critic Algorithms and Applications, 2019