Project 3 : Continuous control

① Detailed explanation of the algorithm I chose to implement.

The goal of the project was to implement an algorithm which would learn to solve the "run" task of the Walker domain from DeepMind Control Suite. It is a task with continuous state and action spaces. Thus, my algorithm of choice was Soft Actor-Critic algorithm, which is a recent algorithm supposed to outperform the state-of-the-art at the time of publishing. First, I will briefly summarize the overall ideas behind (soft) actor-critics. Secondly, I will describe my implementation of Soft Actor-Critic.

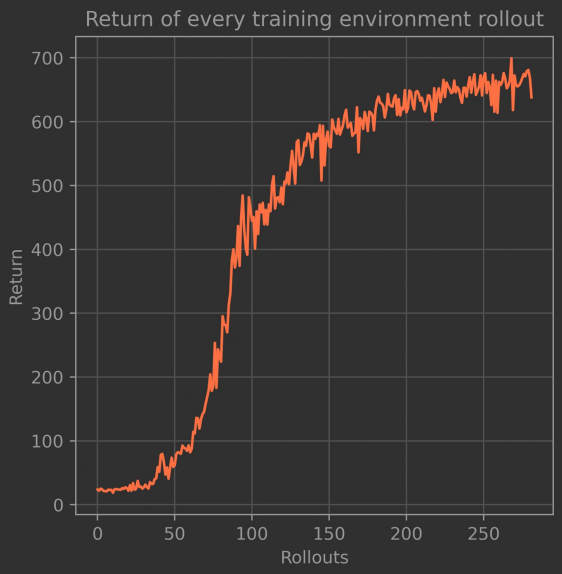
The main idea behind actor-critic algorithms is that we want to optimize policy directly. In previous methods, this was done by optimizing either state value function or state-action value function and the policy was then extracted from the values. In these methods we want to skip the intermediate and optimize policy, also because it is often continuous and stochastic. There are algorithms utilizing policy gradients to optimize only policy without anything else. One of these algorithms is REINFORCE and because it uses Monte-Carlo estimates for the policy gradient, it can suffer from high variance. Actor-critic methods try to mitigate this issue of high variance by learning both policy and one or more value functions. From this comes the name of this class of algorithms, where the learned policy is called actor and the learned value function is called critic. In this sense the critic corrects the actor to reduce variance. It suggest the amount by which the actor should update. These methods can be used for both discrete and continuous state and action spaces. The Walker domain from DeepMind Control Suite has both state and action spaces continuous. Soft Actor-Critic is an extension by Maximum Entropy RL. This theory adds an entropy term $H(\pi(\cdot|s))$ into the policy loss function which helps the agent to explore more and to avoid getting stuck in local optima. Because the task is continuous, both the critic Q functions and actor policy π have to be approximated, in this case by neural networks. Both networks are optimized by stochastic gradient descent on samples drawn from some replay buffer of past transitions. The original paper includes optimizing policy $\pi(a|s)$, state-action value function $Q(s,a)$ and also a state value function $V(s)$. In a more recent paper, they remade the updates in such a way that $V(s)$ is no longer necessary. The algorith also uses a few tricks to improve performance and stability similarly to previous algorithms. It uses two Q functions as in Double Q-learning to remove overestimation bias and also they use target networks which are not learned but their weights are updated from the main two Q-functions either periodically after some number of iterations or as an exponentially weighted average of the past network parameters.

In my implementation of Soft Actor-Critic I chose to represent the policy and Q-functions as neural networks. Q-function architecture consists of two hidden layers of sizes corresponding to hyperparameter `q-hidden`, which is by default 256 neurons per layer. Every layer except the output layer is followed by ReLU nonlinearity. Q-function has two inputs, state and action, so these two vectors are concatenated into one long vector which is then passed to the neural network. Output is a single number $Q(s,a)$. Policy is also realized by a neural network but because it it is continuous and stochastic it is done in a different way. This time, only state vector is the input of the network and again it has two hidden layers. Their size is determined by hyperparameter `pi-hidden` and the default value is also 256. In contrast to Q-functions, the nonlinearity can be selected as ReLU or tanh by hyperparameter `pi-nonlinearity`. Output of the network is location and scale of a Gaussian distribution. The action to be played is sampled from this Gaussian. Moreover, because the Normal distribution is unbounded, the output is squashed to the interval $(-1,1)$ by tanh function and scaled to fit the action space of the task. Both main Q-functions and the policy networks are optimized by ADAM optimizer with learning rate set by hyperparameter `learning-rate`. The target networks have initially the same weights as the main networks. They are then updated from the main Q-functions as an exponentially weighted average $\bar{\theta}_i = \tau\theta_i + (1-\tau)\bar{\theta}_i$ where $i \in \{1,2\}$ denotes one of the Double Q-learning functions $\bar{\theta}_i$ are weights of the target networks, θ_i are weights of the main Q-functions and τ is a hyperparameter `tau`. The algorithm runs in episodes (number given by `num-episodes`) and each episode has a number of steps given by `steps-per-episode`. In each step, an action is sampled. For the first `initial-steps` steps the actions are sampled uniformly from the domain action space, after that the are sampled from the learned policy π as described above. This should help exploration as we are not biased by the shape of a Gaussian but every action has the same probability of selection. Then, the sampled action is applied to the environment and a reward, next state and termination signal are observed from the environment. The transition (state, action, reward, next state, done signal) is inserted into the experience replay buffer for future use. The replay buffer has a fixed maximum size given by `replay-size` hyperparameter. Now, if the next state is terminal (every rollout of the environment is 1000 steps long), we just reset the environment and save aggregated reward. If the buffer has enough transition tuples to sample batch of `batch-size` and it is time to update weights (network updates occur once per `update-every` steps and they occur `update-every` times when the situation occurs), we perform the updates. At the end of each episode, `test-num` of tests are conducted with current policy on an independent test environment and the returns are stored. Every weight update begins by drawing a batch B of `batch-size` transitions from the experience buffer. Then, the main Q functions are updated. New actions a' are sampled from $\pi(\cdot|s')$ and $\log\pi(a'|s')$ is computed. From this we compute targets $y = r + \gamma\cdot(1-d)\cdot\left[\min_{i=1,2} Q_i^{target}(s',a') - \alpha\cdot\log\pi(a'|s')\right]$ where γ is discount factor `gamma`, d is the terminal signal and α is a `temperature` parameter, which will be described later. For both Q-functions we compute loss $L_i = \frac{1}{|B|}\sum_{(s,a,r,s',d)\in B}\left(Q_i(s,a) - y(s',r,d)\right)^2$, we compute its gradient by backpropagation and make optimizer step. Similarly, for policy π we compute $L_\pi = \frac{1}{|B|}\sum_{(s,a,r,s',d)\in B}\left(\alpha\cdot\log\pi(\tilde{a},s) - \min_{i=1,2} Q_i(s,\tilde{a})\right)$ and do the same as with Q-functions. The last thing to describe is the `temperature` α. It can be either fixed value (`temperature` $\in\mathbb{R}$) or it can be learned as well (`temperature = NONE`). When learned, it is a single parameter optimized by ADAM with `learning-rate`. Its loss is $L_\alpha = \frac{1}{|B|}\sum_{(s,a,r,s',d)\in B} -\alpha\left[\log\pi(\tilde{a},s) + \bar{H}\right]$, where $\tilde{a}$ is sampled from π, not taken from batch, and $\bar{H}$ is an entropy parameter corresponding to $-\dim(A)$ where A is action space of the domain.

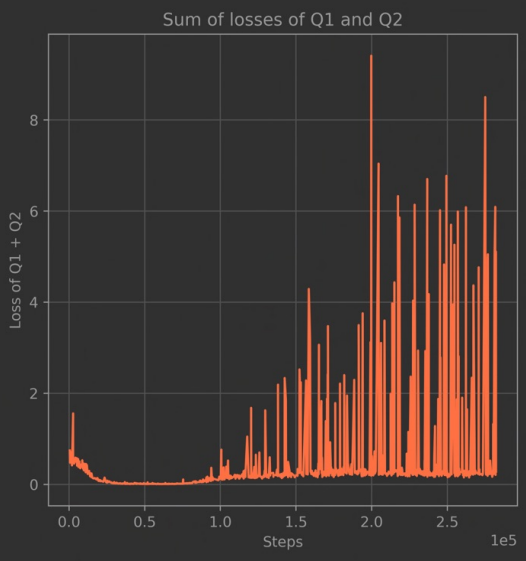
Training results

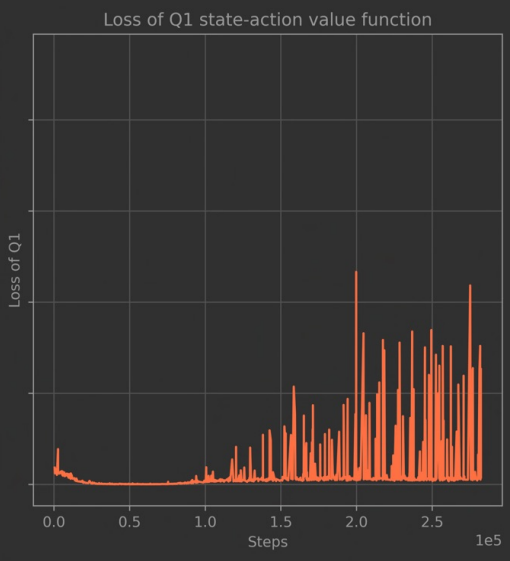
I have run multiple experiments with different hyperparameter settings. However not that many combinations showed good results. For example, the temperature parameter α is very sensitive and fixed values did not work at all. The same happened with the update of target Q networks. In the original paper, they showed that the weights can be copied every 1000 update steps instead of updating in every iteration as an exponentially weighted average. In my implementation it did not work either.

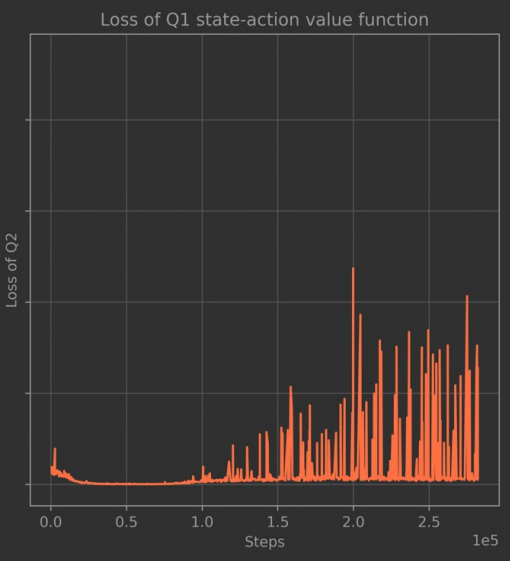
Here, I will show and comment my best results. Note, that I terminated learning maybe a bit sooner than necessary. I have achieved cumulative reward close to 700 and it had already run a long time. So even though the algorithm could have improved more, I wanted to free resources for others as my results were not improving as much.



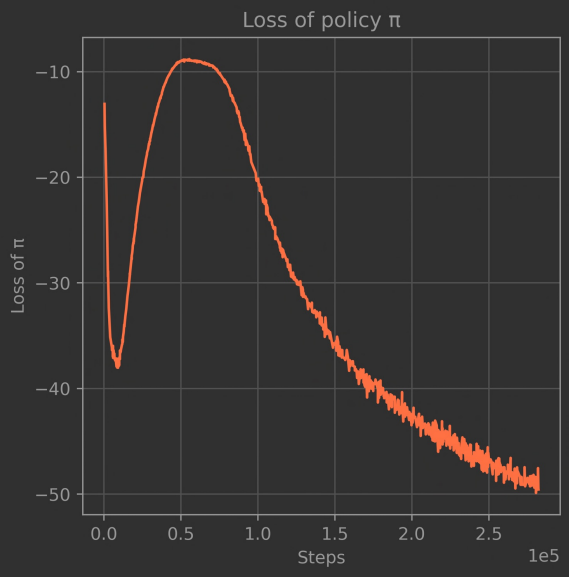
As I mentioned above, the algorithm has not converged yet, but we already can see a huge slow-down in increasing returns. The chart on the left shows cumulative undiscounted return for each environment rollout (i.e. run). As can be seen from the x axis, the algorithm did close to 300 simulations in the environment. There was a rapid growth of return in the middle hundred simulations, the start and end were slower. On the right side, these are displayed average test returns. After every 10 rollouts for learning there were 10 runs with just policy to verify progress. Tests were run in a separate environment with random seeds. There were two kinds of tests. Deterministic test did not sample from the Gaussian distribution but just played the mean action. The stochastic one sampled, on the other hand. Their outcomes are very close, which can mean that the scale of the distribution is not very large. Also, the shape and values correspond to the learning curve, so the algorithm works nicely.

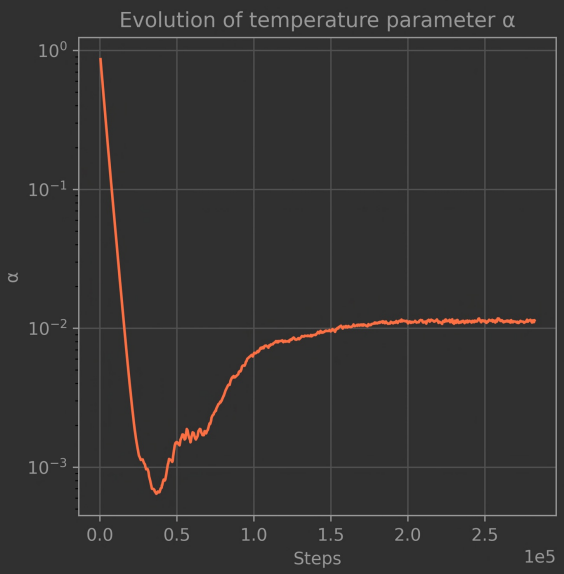






The Q-function losses are not very interesting as it converged really quickly close to zero. Later, we can see some instabilities which can correspond to exploration of new actions. One important thing is the x axis. My version of Soft Actor-Critic did update of weights on every step of the environment. Except in the beginning, when the experience replay buffer was too empty for sampling. Thus considering almost 300 rollouts where every rollout is 1000 steps long, we arrive at some 300 000 updates of the weights. In the paper, the numbers were bigger by an order of magnitude (millions of steps). Maybe my algorithm could have reached the same results if it had run for longer.





These two charts are way more interesting. Both policy π and temperature parameter α were updated on every step of the environment. Thus x axis runs again from 0 to almost $3 \cdot 10^5$. The graph on the left shows loss of policy π computed as mentioned in the first section. First, we see rapid decrease, then rapid increase back to a similar value. After, there is continuous and steady decrease, which it seems would have continued. I have no concrete explanation for the spike but I noticed that it happens at the same time as Q-function losses Y are very close to zero. When the Q-function loss starts to show instabilities and rise a little bit the policy loss starts to drop again. It was quite early, so my guess would be that as the networks are learning from each other, they managed to get out of some local optimum. The chart on the right shows the value of temperature parameter α, which stands in many update formulas in the first section. Note that this is the only graph that has logarithmic scale on y axis to better show the learning curve. As I mentioned before, setting fixed value of α didn't show good results and this is a proof why. The recommended value for α is 0.2 but as can be seen from the graph, the learned value is an order of magnitude smaller. Also, it makes sense to learn the parameter as it influences exploration and in general, we want to explore less in the later stages.

In conclusion, my algorithm was able to learn the walker domain successfully. When visualized, the agent was able to run the whole 10^3 steps without fall and without noticeable staggering. Evaluating my best policy by the provided script with seed 0 leads to value 691.66 which is also a good result for me. The agent for evaluation uses deterministic version of the learned policy, meaning that it returns mean of the Gaussian and does not sample from it. That was also the deterministic test above.

Hyperparameters

- discount factor $\gamma = 0.99$
- weight for the exponentially weighted average $\tau = 0.05$
  - the target Q-networks are updated every 1 step
- learning rate for all optimizers (Q_1, Q_2, π, α) is $3 \cdot 10^{-4}$
- the weights are updated every 1 step
- ran with 1000 episodes each consisting of 10 000 steps (10 full rollouts) → however, only 30 episodes were completed before shutting it down
- random seed for numpy, torch and the environment is 42
- temperature α learns on its own
- replay buffer size is 10^6 transitions
- batch size for sampling from replay buffer is 256
- all networks have 256 neurons per hidden layer
- all networks use ReLU as activation between layers
- at the end of every episode 10 tests were run
- initial 10^3 steps are uniformly sampled instead of sampling the policy