Reinforcement Learning Seminar
Project (fundamental version)

The Fundamental project focuses on the Agent side of a Reinforcement Learning setting. In this project, you have to implement a Reinforcement Learning agent.

Policy Optimization

The objective of Reinforcement Learning is to compute a policy π that maximizes the expected discounted return it obtains: π maximizes the expected (over episodes) sum (over time-steps) of $\gamma^t r_t$, with r_t the reward obtained at time t.

Approaches such as Policy Gradient, seen during the lectures, consider that the policy π is parametric. The action it produces depends on the state and some learnable parameters, such as the weights of a neural network. Policy Gradient finds the parameters that maximize the Policy Gradient objective, $R_t \log \pi(a_t \mid s_t)$. This is the sum of rewards obtained from time-step t to the end of the episode, times the probability that the policy associates with the action a_t in s_t.

Policy Gradient uses Gradient Descent to update the policy: the policy is executed for one episode in the environment, then the returns are computed for every action that was executed in every state visited during the episode. This leads to the production of a bunch of R_t, and their corresponding s_t and a_t. Then, a neural network, that represents the policy, is "forwarded" to obtain $\log \pi(a_t \mid s_t)$. The Policy Gradient loss is computed, and its gradient is computed. The weights are then moved in the direction of the gradient by a small learning rate.

The problem with Policy Gradient is that it follows a gradient. A gradient has only local information about how the parameters influence the policy. Gradient descent suffers from problems, such as getting stuck in local optima. Choosing the learning rate is also difficult. As such, gradient-free approaches, that don't use a gradient, can become interesting.

Gradient-Free Optimization

Gradient-Free optimization methods are able to optimize a function without computing its gradient. In Reinforcement Learning, this means that they allow to improve a policy without having to compute the gradient of its parameters.

The core of Gradient-Free methods is to consider the parameters θ of the policy, and "perturb" them. So, random values are added to them. The resulting perturbed parameters are then evaluated in the environment, and used to compute updated (hopefully better) parameters.

Zeroth-order Optimization

Zeroth-order optimization perturbs θ in two opposite directions, evaluates the two perturbations, and uses which one achieves the best results in order to compute how to move θ so that the quality of the policy improves:

1. The policy π has parameters θ. The parameters are, in practice, Numpy arrays (in PyTorch, model.parameters() allows to get them).
2. Produce one perturbation vector of the same shape as θ, for instance by using Numpy.randn_like. This perturbation vector consists of random values, having an average of 0 but some variance. So, some values are positive, some are negative.
3. Produce 2 perturbations of θ, using the perturbation vector. The 2 perturbations are θ+ and θ-. The + perturbation is simply the perturbation vector as is, while the - perturbation is minus the perturbation vector.
4. Evaluate these 2 perturbations. For this, the perturbed θ values can be "stuffed back into" the policy π, that can be executed for one episode in the environment. It can also be interesting to evaluate each perturbation by performing several episodes, and averaging their return. This leads to an evaluation that is less dependent on randomness in the environment.
5. θ+ and θ- are now associated with a score, the result of the evaluation.
6. A sort of gradient of θ, that we did not have to compute, is now given by "0.5 × (score of θ+ - score of θ-) × θ+".
7. Move θ, the parameters of the policy, in the direction of the gradient. For instance, "θ += 0.001 * gradient".
8. Go back to Step 1.

Population Methods

Zeroth-order optimization still depends on a learning rate: once the perturbations are evaluated, they are used to compute a direction in which to change θ, and the direction is followed with a small learning rate. This still has problems with local minima.

Population methods follow a simpler approach: many perturbations of θ are produced, and the best one is copied into θ:

1. The policy π has parameters θ.
2. Produce N perturbations of θ, now called θi with i going from 1 to N.
3. Evaluate each θi in the environment, by performing one or more episodes (as with Zeroth-order optimization). This produces one "score" per θi.
4. Select the θi that has the largest score, and put it in θ. The policy has now been updated.
5. Go back to Step 1.

Population methods can be made more complex (selecting the top-K perturbations and doing something on them, for instance), but the simple method described above is already enough for this project.

Population methods are not sensitive to local optima, and are generally better at eventually learning the optimal policy. With an infinite amount of time, they will try every possible θ, and find the one that leads to the best returns. However, population methods are not sample-efficient. Each application of the algorithm above requires at least N full episodes in the environment, to compute only a single updated θ.

It is therefore not clear whether zeroth-order methods or population-based methods are preferrable.

The project

For this project, your tasks are as follows:

1. Implement a parametric policy, with parameters θ. We recommend that you use PyTorch and a torch.nn.Module as the policy. The module takes as input the state of the environment, and produces an action as output. We suggest that it contains a single hidden layer of 128 neuros.
2. Consider the LunarLanderContinuous task in the OpenAI Gym. You can use any version of the Gym or Gymnasium for this (all are available in Pip). It has a simple continuous state (8 floats), and a simple continuous action (2 floats). The policy is now a neural network with 8 inputs, 128 hidden neurons, and 2 outputs. It ".parameters()" attribute allows to get its parameters, as a list of PyTorch tensors (so, this is a list of tensors, not one single big tensor).
3. Implement the simple Population Method described above to learn a policy in LunarLanderContinuous.
4. Implement the Zeroth-order method described above, in a separate file.
5. For both methods, produce a learning curve for LunarLanderContinuous. The X axis is the total number of episodes that were executed in the environment (so, not the number of applications of the algorithm, but every single episode that was used for evaluation), the Y axis is the return obtained during the episode. We recommend that you simply "print('RETURN', episode_number, return, file=some_file)". This produces a log file that contains episodes numbers and returns, and can be processed by "gnuplot". This approach is simpler than using Tensorboard, and more robust than Matplotlib (with Matplotlib, if your agent runs for 2 hours and then the plot does not look nice, you have to re-run the agent for 2 hours to get a new plot).