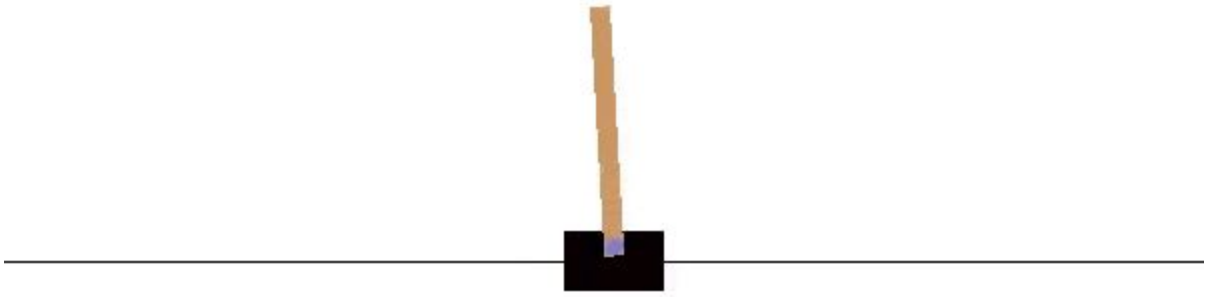
REINFORCEMENT LEARNING (PART 3)

Prepared by Rubén Martínez Cantín rmcantin@unizar.es

In this project, you will implement a policy search algorithm to control a cart and pole scenario using the OpenAI Gym.



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 episode also ends after 200 timesteps for CartPole-v0 and 500 timesteps for CartPole-v1.

The code for this project is a single script file that can be downloaded here.

Environment Description

(adapted from https://github.com/openai/gym/wiki/CartPole-v0)

States

4 continuous values

Num	State name	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Actions

2 discrete actions

Num	Action
0	Push cart to the left
1	Push cart to the right

Note: The amount the velocity is reduced or increased is not fixed as it depends on the angle the pole is pointing. This is because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it

Reward

Reward is 1 for every step taken, including the termination step.

Starting State

All states are assigned a uniform random value between ±0.05.

Episode Termination

1. Pole Angle is more than ±12°
2. Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)
3. Episode length is greater than 200 (500 for v1).

The main class that we are going to use is the `CartPole` class, which defines the environment and the agent being used in the constructor (`__init__`). Feel free to change those.

For development and debugging, it is useful to start with `CartPole-v0` because the episodes are shorter (200 timesteps). However, the system should be able to work on `CartPole-v1` (500 timesteps).

Question 1: Random and Naive Agents

Before doing any RL, we are going to get familiar with the environment. First, you can use the `NaiveAgent`, which is already implemented. You can change the agent being used by changing the constructor (`__init__`) of the `CartPole` class. There, you can also change the environment used. You can try similar environments from OpenAI Gym.

Note that `NaiveAgent` does not learn and its policy is very simple, but it is better than not doing anything.

- Run `CartPole-v0` and figure out what the policy is doing.

This is called *bang-bang control*, and it is the simplest controller that can be designed. This is the kind of controller that a thermostat uses.

Next, you should implement a random policy. The policy should behave randomly independently of the state of the system. This is the kind of policy that should be called for exploration if we used an epsilon-greedy policy.

- Implement the policy method for the `RandomAgent` class.

Note that you don't need to implement the rest of the methods because `RandomAgent` does not learn.

The NaiveAgent should be able to outperform the RandomAgent, even with such simple policy.

Question 2: Linear Agent

Now we are going to implement our first agent using policy gradient. You need to complete the methods:

- __init__
- policy
- policy_gradient
- update_policy
- updale_learning_rate

First, we need to initialize the parameter vector for the policy in the __init__ method. In this case, we cannot initialize everything to 0. A good strategy is to use a random value. Because we might need positive a negative values, the weights might be samples from a normal distribution (see numpy.random.randn).

You can also include any other initialization parameter.

Policy:

We are going to use a softmax policy. Remember that the equation is

$$\pi_\theta(a|x) = \frac{e^{\phi(x,a)^T\theta}}{\sum_b e^{\phi(x,b)^T\theta}}$$

In this case, the policy should return 2 probability values. One for the action = left, and the other for action = right.

The features are directly the state times an indicator function for the action. Another way to see the features is that the policy is actually the combination of two different functions: one for the probability of the *left* action and one for the probability of the *right* action, each with a different set of parameters.

That is, the policy can be written as:

$$p(a_t = left | x_t = X) \propto e^{X_0\theta_0 + ... + X_4\theta_4}$$
$$p(a_t = right | x_t = X) \propto e^{X_0\theta_5 + ... + X_4\theta_8}$$

where X is the values of the state vector at that moment. As you can see, with this policy, the number of parameters is *n_states * n_actions*. Remember that the $\propto$ symbol means proportional, because we are computing only the numerator of the policy. Then, we sum all the probabilities to get the denominator and normalize all the values. In the final vector, you should have that p(a=left) + p(a=right) = 1.

Gradient-Log:

Remember that the gradient-log of the softmax policy is:

$$\nabla_\theta \log \pi_\theta(a|x) = \phi(x, a) - \sum_b \pi_\theta(b|x)\phi(x, b)$$

In this case, the gradient should be computing considering the actual **action that has been taken**. Note that the action taken might not be the one with the highest probability. This gradient should be a vector with a partial derivative for each one of the parameters. **Important**: if a parameter is not involved in an action, that feature is 0. For example, these are some values of the gradient (we have removed the xt = X for clarity).

$$\frac{\partial \log p(a_t = left)}{\partial \theta_0} = X_0 - (X_0 p(a_t = left) + 0 p(a_t = right))$$
$$\frac{\partial \log p(a_t = left)}{\partial \theta_5} = 0 - (0 p(a_t = left) + X_0 p(a_t = right))$$
$$\frac{\partial \log p(a_t = right)}{\partial \theta_0} = 0 - (X_0 p(a_t = left) + 0 p(a_t = right))$$

IMPORTANT: Gradient checking

You might one to check the values of your gradient for debugging. For each parameter, you can perturb its value with a small value epsilon and compute the finite difference gradient approximation:

$$\frac{\partial \log p(a_t = left)}{\partial \theta_0} \approx \frac{\log \pi_\theta(a_t = left|\theta_0 + \epsilon) - \log \pi_\theta(a_t = left|\theta_0 - \epsilon)}{2\epsilon}$$

You can use `numpy.isclose` and `numpy.allclose` to compare similar values.

Policy update:

In the `run_episode` method, you can check how the simulator follow the policy and collect a list of gradients and rewards for each time step. Now, it is your turn to fill the `update_policy` method and update the policy parameters with those lists.

You should follow the REINFORCE algorithm:

for each step t in episode **do**:

$$U_t \leftarrow \sum_{i=t+1}^{T} R_t$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|x_t) U_t$$

Note that, the list of gradients and rewards should be the same size as the number of timesteps of the episode (for example: 200). Then, you should update the parameters of the policy also 200 times, before running the next episode.

Important: do not worry about efficiency in this part. Grading would be based on the accuracy of the results and the technical quality of the report.

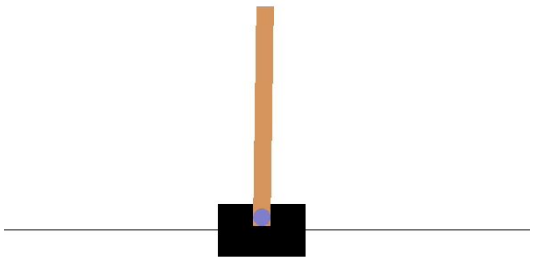
Update learning rate (OPTIONAL):

You can achieve a decent performance with a constant learning rate. However, it might be tricky to find a good value. Instead, we are going to use a decaying sequence, that is, we make alpha with a smaller value after each episode.

The simplest decaying alpha = 1/k, where k is the number of episodes that have run. However, the sequence starts {1, 0.5, 0.33, 0.25…} which are quite large values.

Instead we can use alpha = 1/(k + c), with a large constant value for c. Then, the initial values of the sequence are smaller, but we keep the decaying effect.

Congratulations! Now you should have something like this:



Grading:

You should send a zip file with your code and a report of the process of development and analysis of the results. The report should complement the code, do not describe it. You might also include a short section with ideas on how to improve the learning process.