**REINFORCE algorithm report**

The following report describes the algorithm “REINFORCE”. The algorithm was invented by Ronald J. Williams in 1992, and the main idea is that during learning that resulted in good outcomes are positively reinforced and should become more probable, while the ones with bad outcome become less probable. The algorithm needs three components: (1) a parametrized policy (2) an objective to be maximized (3) a method for updating the policy parameters.

from torch.distributions import Categorical

import gym

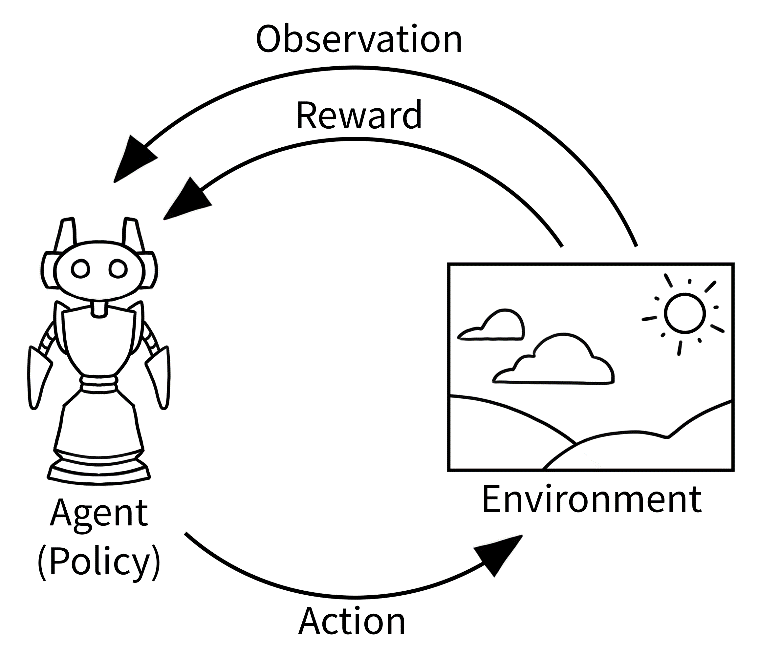
import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

These lines import all the necessary packages to use. Categorical “Creates a categorical distribution parameterized by either probs or logits (but not both)”, where the samples are integers from 0 to K-1. Gym is an API for reinforcement learning and a diverse collection of reference environments, and it implements an angent-environment loop as seen in the following picture.



<https://www.gymlibrary.ml/content/api/>

Numpy is a well-known package for scientific computing, including diverse mathematical functions and linear algebra routines. Torch package “contains data structures for multi-dimensional tensors and defines mathematical operations over these tensors”. Torch nn refers to neural networks and optim to optimization algorithms.

gamma = 0.99

Gamma α is the learning rate. It controls the size of the parameters update.

class Pi(nn.Module):

In this class is where the policy network. Policy π maps states to action probabilities, how an agent produces actions in the environment to maximize the objective. In REINFORCE algorithm, an agent learns a policy and uses this to act in an environment. The policy is in charge of maximizing the cumulative discounted rewards.

def \_\_init\_\_(self, in\_dim, out\_dim):

        super(Pi, self).\_\_init\_\_()

        layers = [

            nn.Linear(in\_dim, 64),

            nn.ReLU(),

            nn.Linear(64, out\_dim)

        ]

        self.model = nn.Sequential(\*layers)

        self.onpolicy\_reset()

        self.train() *# set the training mode*

In these lines we are constructing the policy network. In this case, it is a one-layer MLP with 64 hidden units, but we could substitute the model. We use the package nn to build it and give it the methods to reset the policy and train.

    def onpolicy\_reset(self):

        self.log\_probs = []

        self.rewards = []

The at the time of policy reset, the logs of the probabilities and the rewards are cleared to prepare for the next state after training, as we cannot reuse the trajectory. **log\_probs** records action probabilities.

    def forward(self, x):

        pdparam = self.model(x)

        return pdparam

The model uses a neural network, which learns from a function. In these lines is where we make happen the sequential computation happens as a forward pass. It will take the input x to produce the predicted outputs.

def act(self, state):

        x = torch.from\_numpy(state.astype(np.float32)) *# to sensor*

        pdparam = self.forward(x) *# forward pass*

        pd = Categorical(logits = pdparam) *# probability distribution*

        action = pd.sample() *# pi (a | s ) in action via pd*

        log\_prob = pd.log\_prob(action) *# log\_prob of pi(a|s)*

        self.log\_probs.append(log\_prob) *# store for training*

        return action.item()

Act method defines the method to produce action. With Categorical, we construct an instance of an action probability distribution.

def train(pi, optimizer):

*# inner gradient-ascent loop of REINFORCE algorithm*

    T = len(pi.rewards)

    rets = np.empty(T, dtype = np.float32)

    future\_ret = 0.0

*# compute the returns efficiently*

    for t in reversed(range(T)):

        future\_ret = pi.rewards[t] + gamma \* future\_ret

        rets[t] = future\_ret

    rets = torch.tensor(rets)

    log\_probs = torch.stack(pi.log\_probs)

    loss = - log\_probs \* rets *# gradient term; negative for maximizing*

    loss = torch.sum(loss)

    optimizer.zero\_grad()

    loss.backward() *# backpropagate, compute gradients*

    optimizer.step() *# gradient-ascent, update the weights*

    return loss

The policy is a very important part in the lines of code. It is given as a parameter along the optimizer, and during the control loop it generates the actions to make it run.

The training step is repeated until the network has converged (outputs stop changing or the loss has minimized).

We have to increase the expected reward. After the time steps the loss will be calculated, and from this loss the gradient is calculated with respect to the parameters of the network. The optimizer is used to update the network parameters using the gradient.

def main():

    env = gym.make('CartPole-v0')

    in\_dim = env.observation\_space.shape[0] *# 4*

    out\_dim = env.action\_space.n *# 2*

    pi = Pi(in\_dim, out\_dim) *# policy pi\_theta for REINFORCE*

    optimizer = optim.Adam(pi.parameters(), lr = 0.01)

This is the first part of the main() method. The environment we are working with currently is the CartPole-v0. We obtain the environment Cart-Pole in the library gym, and it consists on simulating a pole attached to a cart as an unstable system. The goal is to move the cart left and right to keep the pole from falling.

There are two four dimensions for the input (state): cart position, cart velocity, pole angle, and pole velocity at tip. For the output there are two, the probability of action left, and the probability of action left. We send these dimensions to our policy.

for epi in range (300):

        state = env.reset()

        for t in range(200): *# cartpole max timestep is 200*

            action = pi.act(state)

            state, reward, done, \_ = env.step(action)

            pi.rewards.append(reward)

            env.render()

            if done:

                break

Those lines are to execute the episodes from our reinforcement learning system. The agent and the environment are interacting and exchanging signals (state, action and reward; an experience). In the current algorithm there are 300 episodes and 200 time steps.

loss = train(pi, optimizer) *# train per episode*

        total\_reward = sum(pi.rewards)

        solved = total\_reward > 195.0

        pi.onpolicy\_reset() *# onpolicy: clear memory after training*

        print(f'Episode {epi}, loss: {loss}, \

        total\_reward: {total\_reward}, solved: {solved}')

if \_\_name\_\_ == '\_\_main\_\_':

    main()