Learning How to Train Robots through Evolutionary and Reinforcement Learning Algorithms

Stefano Nolfi
CNR-ISTC, Roma, Italy
stefano.nolfi@istc.cnr.it

Abstract

This paper explains how to train robots in simulation through evolutionary and reinforcement learning methods. The objective is that to enable the reader to acquire hand-on experience on this topic and familiarize with state-of-the-art algorithms and software tools. More specifically the goal is that to enable the reader to understand how to implement a neural network policy, how to implement an evolutionary algorithm, how to use the Open-AI Gym environments, how to create new environments, how to train robots with state-of-the-art evolutionary and reinforcement learning algorithms.

1. Introduction

This paper explains how to train robots in simulation through evolutionary and reinforcement learning methods. The objective is that to enable the reader to acquire hand-on experience on this topic and familiarize with state-of-the-art algorithms and software tools that are used to carry on research in evolutionary robotics and in reinforcement learning robotics.

The course does not require particular pre-requisite beside a knowledge of the Python programming language. A basic knowledge of C++ is also recommended. For background knowledge on robotics and machine learning methods applied to robotics, the associated book from the same author entitled "Behavioral and Cognitive Robotics: An Adaptive Perspective" constitutes an ideal companion (forthcoming).

The course includes 7 exercises that can be completed in about 12 hours. All the software required is available ready-to-use through the Docker container prepared by Vladislav Kurenkov (see https://github.com/snolfi/evorobotpy). Alternatively, the reader can install the software packages required by following the instructions included in the following sections.

2. AI Gym

In this first section we will familiarize with AI Gym and we will learn how to implement a neural network policy and an evolutionary strategy from scratch. The code developed will then be used to solve some of problem included in the AI Gym library.

Gym is a free toolkit developed by Open-AI (gym.openai.com, Brockman et. al, 2016) for testing and comparing machine learning algorithms applied to agents situated in an external environment. The tool includes components for simulating a wide range of test problems. The implementation of the environments is standardized so to enable the user to apply the algorithm of interest to a wide range of problems without the need of customization.

Gym include a set of simple classic control problems (the cart-pole problem, the pendulum, the acrobat, ext.), the MuJoCo control problems that involve swimming and walking robots with varying morphologies, a series of robotic environment based on the Fetch and ShadowHand robots, and the Atari games. These tools are provided as free software with the exception of the MuJoCo and of the

robotic environments that depends on the MuJoCo commercial simulator (free one-year licenses are available for students). Other environmental problems, compliant with the AI Gym standard, have been developed from third party. In particular, a good free alternative to the MuJoCo environments is provided by PyBullet (see Section 3.3) that includes alternative and more realistic implementations of the locomotor problems and other interesting environments.

To start, download, build and run the container including all software mentioned in this document by following the instruction included in https://github.com/snolfi/evorobotpy. Alternatively install the AU Gym tool following the instruction included in the https://gym.openai.com/docs/ page.

The fundamental commands available in Gym are summarized in the following python scripts that creates a cart pole balancing problem, resets the position of the cart, makes the cart move for 200 steps by setting the state of the actuator randomly, and renders graphically the behavior of the agent.

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(200):
    env.render()
    observation, reward, done, info = env.step(env.action_space.sample())
env.close()
```

The CartPole-v0 environment consists of a cart that can move along the x axes with a pole attached on the top through a passive joint. The task of the agent consists in controlling the actuator of the cart, that can apply a positive or negative force along the x axis, so to maintain the pole balanced. The gym.make() function is used to create the environment.

The reset() method initializes the position of the agent and of the environment. This function is usually called at the beginning of evaluation episodes. In the case of the CartPole-v0, it places the cart in the center of the x axis and the pole in the vertical position inclined by a small random angle.

The render() method displays graphically the agent and the environment.

The step() method receives as input the desired state of the actuator, updates the actuators, performs a simulation step, computes the reward for the step, and checks the conditions for terminating the episode. It returns the observation, i.e. the updated state of the agent sensors, the reward, a Boolean value that indicate whether the episode should terminate, and a dictionary that might contain additional information. In the case of the CartPole-v0 the observation includes the position and the velocity of the cart along the x axes, the angle of the pole, and the velocity of the tip of the pole. The action includes an integer value in the range [1, 2] that specifies whether the actuator push the cart left or right. The action_space.sample() method return a random action or a vector of random actions suitable for the current environment.

To know the type and the dimensionality of the observation and action space, you can print the observation_space and action_space variables as shown in the script below. The object Box indicates a vector of real values and the first element of the object indicates the number of elements forming the vector. The object Discrete indicates integer positive values, and the first element of the object indicates the range of the integer values. You can also inspect the minimal and maximum values of each observation and action elements by printing the observation_space.high, observation_space.low, action_space.high and action_space.low variables.

```
import gym
env = gym.make('CartPole-v0')
print(env.action_space)
```

```
#> Discrete(2)
print(env.observation_space)
#> Box(4,)
print(env.observation_space.high)
#> array([ 2.4       ,          inf,  0.20943951,          inf])
print(env.observation_space.low)
#> array([-2.4       ,         -inf, -0.20943951,         -inf])
```

Exercise 1. Familiarize with these commands and with the other environments available in gym. Exploit the possibility to issue the commands line by line on the python interpreter and to print variables to familiarize with the structure of the data. From the environments page of the gym.openai.com site you can also inspect the source code of the some of these environments starting from the simple classic control problems (e.g. CartPole-v0, pendulum-v1 ext.)

2.1 Implement your first neural network policy and training algorithm

This section includes a guideline for implementing from scratch a neural network policy for Gym problems, e.g. for the CartPole-v0 problem (Exercise 2a), and an Evolutionary Strategy for training the policy (Exercise 2b).

The policy consists of feed-forward neural network with a layer of sensory neurons encoding the state of the observation, a layer of internal neurons, and a layer of motor neurons that encode the action state. The activation of the internal and motor neurons is computed with the tanh function. We assume that observation is constituted by a vector of real numbers (4 in the case of the CartPole-v0) and that action is constituted by a vector of real numbers or by an integer in the range [1, amax]. In the latter case, the network includes amax output neurons and the action is set to the index of the most activated output neuron. In the CartPole-v0 action is constituted by an integer in the range [1,2] and consequently the network should include two motor neurons.

Exercise 2a. Implement a neural network controller for a Gym problem (e.g. the CartPole-v0 problem). Initialize the network with random parameters, and evaluate the neuro-agent for 10 evaluation episodes each lasting 200 steps.

You can use the following code to allocate and initialize the connection weights and the biases parameters.

```
import gym
import numpy as np
env = gym.make('CartPole-v0')

pvariance = 0.1     # variance of initial parameters
ppvariance = 0.02   # variance of perturbations
nhiddens = 5        # number of hidden neurons
# the number of inputs and outputs depends on the problem
# we assume that observations consist of vectors of continuous value
# and that actions can be vectors of continuous values or discrete actions
ninputs = env.observation_space.shape[0]
if (isinstance(env.action_space, gym.spaces.box.Box)):
    noutputs = env.action_space.shape[0]
else:
    noutputs = env.action_space.n

# initialize the training parameters randomly by using a gaussian distribution with
average 0.0 and variance 0.1
# biases (thresholds) are initialized to 0.0
W1 = np.random.randn(nhiddens,ninputs) * pvariance      # first layer
W2 = np.random.randn(noutputs, nhiddens) * pvariance    # second layer
b1 = np.zeros(shape=(nhiddens, 1))                      # bias first layer
b2 = np.zeros(shape=(noutputs, 1))                      # bias second layer
```

During each step of each episode, the activation of the neural network and the state of the actuator should be updated. You can start from the following code to implement the method that update of the activation of the neural network.

```
# convert the observation array into a matrix with 1 column and ninputs rows
observation.resize(ninputs,1)
# compute the netinput of the first layer of neurons
Z1 = np.dot(W1, observation) + b1
# compute the activation of the first layer of neurons with the tanh function
A1 = np.tanh(Z1)
# compute the netinput of the second layer of neurons
Z2 = np.dot(W2, A1) + b2
# compute the activation of the second layer of neurons with the tanh function
A2 = np.tanh(Z2)
# if actions are discrete we select the action corresponding to the most activated unit
if (isinstance(env.action_space, gym.spaces.box.Box)):
    action = A2
else:
    action = np.argmax(A2)
```

Evaluate the agent for multiple evaluation episodes (e.g. 10 evaluation episodes) each lasting up to 200 steps by summing the fitness collected during all episodes. Since the CartPole-v0 problem returns a reward of 1 for each step in which the pole is balanced, the fitness will correspond to the total number of steps in which the agent manages to keep the pole balanced. Clearly, the agent will not be able to balance the pole for many steps by using a policy with random parameters.

Now implement the steady state evolutionary strategy (Pagliuca, Milano and Nolfi, 2018) illustrated in the pseudo-code below and use it to train the parameters of the policy. The procedure starts by initializing the parameters of the population randomly (line 1). The parameters of each individual forming the population is composed by a vector of real numbers of length p where p corresponds to the number of connection weights and biases of the robot's neural network. The parameters of the population (θ) thus correspond to a matrix including λ vectors of length p. Then, for a certain number of generations, the algorithm evaluates the fitness of the individuals forming the population (line 3), ranks the individual of the population on the basis of their fitness (line 5), and then replace the parameters of the worse $\frac{\lambda}{2}$ individuals with a copy with variations of the $\frac{\lambda}{2}$ fittest individuals (line 8). Variations consists of a vector of random numbers generated with a normal distribution with average 0 and variance σ. The evaluation of an individual (line 4) is realized by creating a policy network with the parameters specified in the θ_i vector, by allowing the robot to interact with its environment for a certain number of episodes, and by measuring the fitness of the robot (i.e. the sum of all collected rewards eventually normalized for the number of episodes).

When the population include only 2 individuals, as in this example, the algorithm is equivalent to a stochastic hill-climber that operates on a single candidate solution by (i) adding random variation to the parameters, and (2) by retaining or discarding the variations depending on whether the addition of variations produce an increase or a decrease of the fitness, respectively.

$\sigma = 0.02$: mutation step size
$\lambda = 2$: population size (should be an even number)
$\theta_{1-\lambda}$: parameters of the population
F: evaluation (fitness) function

1 initialize θ_λ :
2 for $g = 1, 2, \ldots$ **do**

3 **for** $i = 1, 2, \ldots \lambda$ **do**
4 evaluate score: $s_i \leftarrow F(\theta_i)$
5 rank individuals by fitness: u = ranks(s)
6 **for** $l = 1, 2, \ldots \frac{\lambda}{2}$ **do**
7 sample noise vector: $\varepsilon \sim N(0, I) * \sigma$
8 $\theta_{u[\frac{\lambda}{2}+l]} = \theta_{u[l]} + \varepsilon$

During the training process you can store in a vector the parameters of the best agent obtained so far. These data can be used to post-evaluate the best individual at the end of the training process for a larger number of evaluation episodes (e.g. 100).

Exercise 2b. Implement the evolutionary strategy described above and use it to train the neural network policy. You can initialize the parameters of the population with random value extracted with a normal distribution with average 0 and standard deviation. Initializing the parameters corresponding to the biases to 0.0 is usually helpful.

Test your program on the CartPole-v0 problem. Does the robot manage to solve the problem? Does it solve the problem every time you run the training process? What happen by changing the parameters (e.g. size of the population, number of hidden units, the variance of the perturbation vector, the number of evaluation episodes). Test your algorithm on other simple control problem such as the Pendulum-v0.

3. Evorobotpy

In this section we introduce evorobotpy (https://github.com/snolfi/evorobotpy) a software library implemented by the author compliant with AI-Gym that permits to evolve robots. Evorobotpy is characterized by simplicity and computational efficiency. Moreover, it includes a series of environments that are qualitatively different from those available in AI-Gym.

Evorobotpy requires python3, the cython, pyglet and matplotlib python packages, a command line C++ compiler, and the GNU Scientific Library available from: https://www.gnu.org/software/gsl.

The /bin folder contains the python scripts that can be used to run and analyze experiments. The /lib folder includes a series of C++ libraries that can be imported in Python. More specifically, a module that permit to create and use neural network policies and a series of environments (see below). The /pybullet folder includes files for customizing the pybullet library and utilities for showing robot morphologies. The /exercises folder includes sample codes that can be used to carry on the exercise described in this document. The /doc folder includes documentation. The folders starting with the letter x include initialization files with suggested parameters for running experiments and pre-evolved robots.

3.1 The net library

Net is a library that permits to create and use neural network policies. The source code (evonet.cpp, evonet.h, utilities.cpp, utilities.h, net.pxd, net.pyx, and setupevonet.py) can be compiled from the ./lib directory with the following command:

```
python3 setupevonet.py build_ext --inplace
cp net*.so ../bin # or cp net*.dll ../bin
```

In case of compilation errors, please check and eventually correct in the file setupevonet.py the directory that include your gsl library and associated include file. The compiled net*.so or net*.dll library should then be copied in the /bin directory.

The usage of the library is summarized in the python script /bin/testnet.py also shown below. The script shows how to import the compiled library, create a recurrent neural network policy, update the network for 10 steps by setting the value of sensory neurons with random values.

To save the computation time required to pass parameters from python to C++, the evorobotpy allocates the vectors of the observation and action in python and passes the pointer of the vector to the C++ library once, when the policy is created. This permits to avoid to pass the vectors every time the network is updated. The same technique is used to speed-up the communication between the training algorithms implemented in python and the environments implemented in in C++, e.g. ErDiscrim, ErDpole, ErPredprey. Overall this permits to exploit the advantages of the python programming language by maintaining an efficiency analogous to a pure C++ application.

The library also permits to create and update multiple networks with identical topologies. A description of the parameters and associated default values can be obtained by executing the /bin/es.py script without parameters.

```
#!/usr/bin/python
import numpy as np
import net
# parameters to be set on the basis of the environment specifications
ninputs = 4            # number of input neurons
noutputs = 2           # number of output neurons
# set configuration parameters
nnetworks = 1          # number of networks
heterogeneous = 0      # whether multiple networks are heterogeneous or not
nlayers = 1            # number of internal layers
nhiddens = 10          # number of hidden units
nhiddens2 = 0          # number of hiddens of the second layer, if any
bias = 1               # whether we have biases
architecture = 2       # full recurrent architecture
afunction = 2          # activation function of internal neurons is tanh
out_type = 3           # activation function of output neurons is linear
winit = 0              # weight initializaton is xavier
clip = 0               # whether the activation of output neuron is clipped in the [-5.0,
5.0] range
normalize = 0          # whether the activation of input is normalized
action_noise = 0       # we do not add noise to the state of output neurons
action_noise_range = 0 # the range of noise added to output units
wrange = 0.0           # range of the initial weights (when winit=2)
nbins = 1              # number of outputs neurons for output values
low = 0.0              # minimun value for output clipping, when clip=1
high = 0.0             # maximum value for output clipping, when clip=1
seed = 101             # random seed
# allocate the array for inputs, outputs, and for neuron activation
inp = np.arange(ninputs, dtype=np.float32)
out = np.arange(noutputs, dtype=np.float32)
nact = np.arange((ninputs + (nhiddens * nlayers) + noutputs), dtype=np.float64)
# create the network
nn  =  net.PyEvonet(nnetworks,  heterogeneous,  ninputs,  nhiddens,  noutputs,  nlayers,
nhiddens2, bias, architecture, afunction, out_type, winit, clip, normalize, action_noise,
action_noise_range, wrange, nbins, low, high)
# allocate an array for network paramaters
nparams = nn.computeParameters()
params = np.arange(nparams, dtype=np.float64)
# allocate the array for input normalization
if (normalize == 1):
    normvector = np.arange(ninputs*2, dtype=np.float64)
else:
```

```
    normvector = None
# pass the pointers of allocated vectors to the C++ library
nn.copyGenotype(params)
nn.copyInput(inp)
nn.copyOutput(out)
nn.copyNeuronact(nact)
if (normalize == 1):
    nn.copyNormalization(normvector)
# You can pass to the C++ library a vector of parameters (weights) with the command:
#nn.copyGenotype(params)  # the type of the vector should be np.float64
# set the seed of the library
nn.seed(seed)
#initialize the parameters randomly
nn.initWeights()
# Reset the activation of neurons to 0.0
nn.resetNet()
# For 10 steps set the input randomly, update the network, and print the output
for step in range(10):
    # generate the observation randomly
    inp = np.random.rand(ninputs).astype(np.float32)
    # pass the pointer to the input vector to the C++ library
    nn.copyInput(inp)
    # update the activation of neurons
    nn.updateNet()
    # print the activation of neurons
    print("step %d ,  print input, output, and input-hidden-output vectors" % (step))
    print(inp)
    print(out)
    print(nact)
```

3.2 The es.py script

The /bin/es.py script permits to evolve and test robots. It uses the net library described above and a series of associated python scripts: /bin/policy.py that includes instructions for creating and updating a network policy, /bin/salimans.py that contains an implementation of the OpenAI-ES algorithm (Salimans et al., 2017), and /bin/evoalgo.py that includes functions for monitoring the evolutionary process, and for saving and loading data.

You can familiarize with the usage of this program by evolving a policy for the acrobot problem. To do that enters into the /xacrobot folder and run the command:

```
python3 ../bin/es.py -f acrobot.ini -s 11
```

The program will start to evolve a policy for the acrobot problem with the program parameters specified in the /xacrobot/acrobot.ini file and the initialization of the seed set to 11. During the evolution process, the program will display every generation the fraction of evaluation steps performed, the best fitness obtained to date, the best fitness obtained during post-evaluation tests to date, the fitness of the best sample, the average fitness of the population, and the average absolute value of parameters. Moreover, during the process the program will save the parameters of the solution that achieved the best fitness during evaluation and post-evaluation episodes (in the file bestSs.npy and bestgSs.npy where s corresponds to the seed indicated with the parameter -s). Moreover, it will save performance data across generations in the file statSs.npy and FitSs.txt.

The program parameter -f is mandatory and should be followed by the name of a text file containing a description of the parameters, i.e. the file acrobot.ini in this case. The parameter -s permit to indicate the number used for initializing the random number generator and to differentiate the file created by different runs. To see an explanation of all available parameters, execute the es.py script without parameters.

The behavior of an evolved agent can be inspected with the command:

```
python3 ../bin/es.py -f acrobot.ini -t bestgS11.npy
```

where the program parameter -t is used to indicate the name of the file containing the evolved solution. If the program parameter -t is not followed by any file name, the program will display the behavior of a robot with random parameters.

To display the variation of performance across generations and the average and standard deviation of performance among multiple runs, you can use the following command, respectively:

```
python3 ../bin/plotstat.py
```

```
python3 ../bin/plotave.py
```

Exercise 3. Run few replications of the experiment by using different seeds (integer numbers). You can use the pre-prepared acrobot.ini file included in the ./xacrobot directory. While the program is running check the source code of the environment available from the https://gym.openai.com/envs/ website to figure out the content of the observation vectors, the content of the action vector, and the way in which the reward is calculated. Plot performance across generations and then observe the behavior of evolved robots.

3.3 Pybullet

Pybullet (https://pybullet.org/; Coumans & Yunfei, 2016-2019) is a powerful dynamical simulator that come as a free software. It provides an interesting series of AI-Gym environments including a series of more realistic implementation of MuJoCo locomotors problems. Moreover, it can be used to carry on experiments with several interesting robotic platforms such as Ghost Minitaur quadruped (https://www.ghostrobotics.io/), KUKA robotics arms (https://kuka.com), the Atlas humanoid robot (https://www.bostondynamics.com/), the iCub Humanoid robot (Sandini et al., 2004), the Baxter robot (https://www.rethinkrobotics.com/), the Franka Emika Panda manipulator (https://www.franka.de/technology), the Cassie biped robot (https://www.agilityrobotics.com/), the Laikago quadruped robot (http://www.unitree.cc/) and the Turtlebot wheeled robot (https://www.turtlebot.com/).

Follow the instruction included in https://pybullet.org/ to install the library.

You can then familiarize with the morphology of the locomotors and with their actuators by running the command:

```
evorobotpy/pybullet/showrobot/python3 showrobot.py
```

run the command once without parameters to see the list of available files and the run the command again by specifying on the available files as parameter.

Observe the behavior displayed by pre-evolved robots included in the evorobotpy/xhopper, evorobotpy/xhalfcheetah, evorobotpy/xant, evorobotpy/xwalker and evorobotpy/xhumanoid folders.

Before evolving new controllers for these locomotor problems with evorobotpy you should overwrite the pybullet files that include the calculation of the rewards with a modified version of the files included in the ./pybullet folder. This since the original rewards function are suitable for reinforcement learning methods but not for evolutionary strategies. To do that identify the pybullet folder that contains the files gym_locomotion_envs.py and robot_locomotors.py, create a copy of the original files, and then overwrite them with the files contained in the evorobotpy/pybullet folder.

Exercise 4. Evolve the robots with the original reward functions and then compare the behavior of robots evolved with the original and revised reward functions, e.g. in the case of the hopper and halfcheetah. Could you explain why the original rewards functions are not suitable for evolutionary strategies ?

3.4 Implementing a new Gym/Bullet environment

This section includes a guideline for creating and using a new Gym environment from scratch. It is an adapted version of a tutorial developed by Backyard Robotics available from https://backyardrobotics.eu/2017/11/27/build-a-balancing-bot-with-openai-gym-pt-i-setting-up/.

This will constitute the **Exercise 5.**

The problem concerns a simple wheeled robot that can be trained for the ability to balance. We can thus name the environment BalancebotBulletEnv-v0 and the folder containing the implementation of the environment balance-bot. The robot will be simulated by using the PyBullet dynamic simulator.

Structuring the balancing bot project

The project requires the following directory and files (for more detailed instructions see OpenAI guidelines):

```
balance-bot/
  README.md
  setup.py
  balance_bot/
    __init__.py
    envs/
      __init__.py
      balancebot_env.py
```

The file setup.py should include a description of the required libraries:

```
from setuptools import setup
setup(name='balance_bot',
      version='0.0.1',
      install_requires=['gym', 'pybullet'])
```

The file __init__.py should include a register instruction with the name of the environment (that can be passed to the function gym.make()) and the entry point of the environmental class in the format {base module name}.envs:{Env subclass name}:

```
from gym.envs.registration import register
register(
    id='balancebot-v0',
    entry_point='balance_bot.envs:BalancebotEnv',)
```

Finally, the file /envs/__init__.py should include the instruction for importing the environment:

```
from balance_bot.envs.balancebot_env import BalancebotEnv
```

Creating the robot

The balancing bot is a robot with two wheels on the same axis that should manage to keep its body balanced, i.e. upright. The robot has three sensors that encode the orientation and angular velocity of its body and the angular velocity of its wheels. Moreover, it has an actuator that controls the angular velocity of the wheels. The balancing bot is made of a rectangular body with two cylindrical wheels on each side.

The physical structure of the robot can be specified in a text file following the URDF (Universal Robot Description Format) used by ROS. A prefilled version of the balancebot_simple.xml file is

included in the /evorobotpy/exercises/ folder and below. Analyze the file to understand how bodies and joints are specified.

```
<?xml version="1.0"?>
<robot name="balance">
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>
  <material name="black">
    <color rgba="0.2 0.2 0.2 1"/>
  </material>
  <link name="torso">
    <visual>
      <geometry>
        <box size="0.2 0.05 0.4"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
      <material name="white"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.2 0.05 0.4"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
    </collision>
    <inertial>
      <mass value="0.8"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"/>
      <origin rpy="0 0 0" xyz="0 0.0 0.3"/>
    </inertial>
  </link>
  <link name="l_wheel">
    <visual>
      <geometry>
        <cylinder length="0.02" radius="0.1"/>
      </geometry>
      <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
      <material name="black"/>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.02" radius="0.1"/>
      </geometry>
      <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
      <contact_coefficients mu="0.8" />
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </inertial>
  </link>
  <link name="r_wheel">
    <visual>
      <geometry>
        <cylinder length="0.02" radius="0.1"/>
      </geometry>
      <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
      <material name="black"/>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.02" radius="0.1"/>
      </geometry>
      <origin rpy="0 1.5707963 0" xyz="0 0 0"/>
      <contact_coefficients mu="0.8" />
```

```
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </inertial>
  </link>
  <joint name="torso_l_wheel" type="continuous">
    <parent link="torso"/>
    <child link="l_wheel"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.0"/>
    <origin rpy="0 0 0" xyz="-0.12 0.0 0.1"/>
  </joint>
  <joint name="torso_r_wheel" type="continuous">
    <parent link="torso"/>
    <child link="r_wheel"/>
    <axis xyz="1 0 0"/>
   <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.0"/>
   <origin rpy="0 0 0" xyz="0.12 0.0 0.1"/>
  </joint>
</robot>
```

Implementing the environment class

The balancebot_env.py file should include the description of the environmental class and of the associated methods: reset(), step(), render(), and seed().

The head of the file should include the instructions for importing of the required components and the definition of the class:

```
import os
import math
import numpy as np

import gym
from gym import spaces
from gym.utils import seeding

import pybullet as p
import pybullet_data

class BalancebotEnv(gym.Env):
    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second' : 50
    }
```

The __init__ method performs some initializations and defines the observation and action space of the robot. The space class permits to define the characteristics of these space. In particular, the spaces.Box() method can be used to specify a continuous space, the dimension of the space, and the minimum and maximum value of each dimension. In this example, the output space includes a single value that indicates the commanded variation of wheel velocity. The observation space contains 3 values that indicate the inclination of the robot, the angular velocity of the robot, and the angular velocity of the wheels, respectively.

```
    def __init__(self, render=False):
        # action encodes the torque applied by the motor of the wheels
        self.action_space = spaces.Box(-1., 1., shape=(1,), dtype='float32')
        # observation encodes pitch, gyro, com.sp.
        self.observation = []
        self.observation_space = spaces.Box(np.array([-math.pi, -math.pi, -5]),
                np.array([math.pi, math.pi, 5]), dtype='float32')
```

```
        self.connectmode = 0
        # starts without graphic by default
        self.physicsClient = p.connect(p.DIRECT)
        # used by loadURDF
        p.setAdditionalSearchPath(pybullet_data.getDataPath())
        self.seed()
```

The seed() method sets the seed of the random number generator that, in turn, influence the initial conditions experienced during evaluation episodes.

```
    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]
```

The reset() method initializes the position, velocity, and orientation of the robot, load the files containing the description of the plane and of the robot, and compute and return the observation. The orientation of the robot is varied randomly within a given range.

```
    def reset(self):
        self.vt = 0      # current velocity pf the wheels
        self.maxV = 24.6 # max lelocity, 235RPM = 24,609142453 rad/sec
        self.envStepCounter = 0
        p.resetSimulation()
        p.setGravity(0, 0,-10) # m/s^2
        p.setTimeStep(0.01)    # the duration of a step in sec
        planeId = p.loadURDF("plane.urdf")
        robotStartPos = [0,0,0.001]
        robotStartOrientation = p.getQuaternionFromEuler([self.np_random.uniform(low=-
0.3, high=0.3),0,0])
        path = os.path.abspath(os.path.dirname(__file__))
        self.botId = p.loadURDF(os.path.join(path, "balancebot_simple.xml"),
                            robotStartPos,
                            robotStartOrientation)
        self.observation = self.compute_observation()
        return np.array(self.observation)
```

The step() method sets the state of the actuators, advances the simulation for a step, computes the new observation, computes the reward, checks whether the condition for terminating the evaluation episode are satisfied or not, and increases the step counter. The method receives as input the action to perform and returns the observation, the reward, a Boolean value that indicates whether the episode should terminate, and an empty dictionary.

```
    def step(self, action):
        self.set_actuator(action)
        p.stepSimulation()
        self.observation = self.compute_observation()
        reward = self.compute_reward()
        done = self.compute_done()
        self.envStepCounter += 1
        status = "Step " + str(self.envStepCounter) + "  Reward " +
'{0:.2f}'.format(reward)
        p.addUserDebugText(status, [0,-1,3], replaceItemUniqueId=1)
        return np.array(self.observation), reward, done, {}
```

The render() method connects the graphic display for the visualization of the robot and of the environment

```
    def render(self, mode='human', close=False):
        if (self.connectmode == 0):
            p.disconnect(self.physicsClient)
            # connect the graphic renderer
```

```
            self.physicsClient = p.connect(p.GUI)
            self.connectmode = 1
        pass
```

Finally, the following four methods: update the desired velocity of the actuators; compute the observation on the basis of the current position, orientation and velocity of the robot; compute the reward; check whether the episode should terminate.

```
    def set_actuator(self, action):
        deltav = action[0]
        vt = np.clip(self.vt + deltav, -self.maxV, self.maxV)
        self.vt = vt
        p.setJointMotorControl2(bodyUniqueId=self.botId,
                                jointIndex=0,
                                controlMode=p.VELOCITY_CONTROL,
                                targetVelocity=vt)
        p.setJointMotorControl2(bodyUniqueId=self.botId,
                                jointIndex=1,
                                controlMode=p.VELOCITY_CONTROL,
                                targetVelocity=-vt)

    def compute_observation(self):
        robotPos, robotOrn = p.getBasePositionAndOrientation(self.botId)
        robotEuler = p.getEulerFromQuaternion(robotOrn)
        linear, angular = p.getBaseVelocity(self.botId)
        return (np.array([robotEuler[0],angular[0],self.vt], dtype='float32'))

    def compute_reward(self):
        # receive a bonus of 1 for balancing and pay a small cost proportional to speed
        return 1.0 - abs(self.vt) * 0.05

    def compute_done(self):
        # episode ends when the barycentre of the robot is too low or after 500 steps
        cubePos, _ = p.getBasePositionAndOrientation(self.botId)
        return cubePos[2] < 0.15 or self.envStepCounter >= 500
```

Installing and using the environment

You can install the environment with the following instructions:

```
cd balance-bot
pip install -e .
```

Finally, you can use balance_bot as a standard gym environment after importing it with the following instructions:

```
import balance_bot
```

For example, you can train the robot with the evorobotpy/bin/es.py script after adding the import statement in the script.

3.5 The ErDiscrim environment

Dynamical simulator, like bullet, are necessary for simulating accurately the dynamics of articulated robots and for simulating the effect of collisions. In the case of wheeled robots moving on a flat surface, however, kinematic simulations can suffice. Kinematic simulations permit to reduce the simulation cost significantly with respect to dynamical simulations. For this reason, evorobotpy

includes a set of environments involving wheeled robots that rely on a kinematic simulations implemented in C++.

One of these environments is ErDiscrim that involves a Khepera robot, i.e. a small wheeled robot provided with infrared sensors (Mondada, Franzi & Ienne, 1993). The robot is situated on a flat arena surrounded by walls and including a cylindrical object. The task to be solved consists in finding and remaining near the cylinder.

The source code (discrim.cpp, discrim.h, utilities.cpp, utilities.h, robot-env.cpp, robot-env.h, ErDiscrim.pxd, ErDiscrim.pyx, and setupErDiscrim.py) can be compiled from the ./lib folder with the following instructions:

```
python3 setupErDiscrim.py build_ext --inplace
cp ErDiscrim*.so ../bin # or cp ErDiscrim*.dll ../bin
```

In case of compilation errors, please check and eventually correct the name of the directory of your GSL library in the file setupevonet.py. The compiled net*.so or net*.dll should then be copied in the evorobotpy/bin directory.

Give a look to the source files to understand how the environment is simulated. The discrim.cpp and discrim.h files include the definition of the gym functions (i.e. env-reset(), env.step(), env.render() ext.) and the calculation of the reward. The robot-env.cpp and robot-env.h files include a series of methods that permit to simulate the translation and rotation movement of the robot in 2D, the activation of the infrared sensors, and the occurrence of collision. The program terminates the evaluation episode when a collision occurs. Consequently, there is no need to model the effect of collisions in detail. The robot-env.cpp and robot-env.h files permit to simulate also two other type of wheeled robots: the ePuck (Mondada et al., 2009) and the MarXbot (Bonani et al., 2010). These robots are used in other environments described below.

Exercise 6. Run 10 replications of the experiment from the evorobotpy/xdiscrim folder by using different seeds. To speed-up the training process you can run 2 instances of the program in parallel or eventually more, depending on the characteristics of your computer. Test and analyze the strategy displayed by best robot of each replication. Describe the strategies of the robots by grouping them in families. Try to explain why the robot of each family behave in that manner. Run other experiments by using a feed-forward neural architecture (without memory). Explain how the behavior of evolved robots differ from those evolved with the LSTM architecture (i.e. the Long Short Term Memory architecture).

3.6 The ErPredprey environment

ErPredPrey is another environment involving two wheeled robots situated in the same environment. More specifically an environment in which a predator and a prey MaxXBot robots play a competing game: the predator robot should catch the prey (i.e. approach and enter in physical contact with the prey), and the prey robot should avoid being caught.

Evorobotpy handle experiments involving two or more robots by using a single observation vector and a single action vector that contains the data of all robots (i.e. the observation of the first robot followed by the observation of the second robot ext.). The length of observation and action vectors is calculated on the basis of the sensory neurons required by each robot and on the basis of the number of robots specified with the parameter *nrobots*. The value of this parameter is passed to the evonet library that create and update the corresponding neural networks. The robots are heterogeneous, as specified by the *heterogeneous* parameters. This implies that each genotype include N**nrobots* genes where N is the number of genes required to encode the parameters of a single network policy. In

experiments in which the heterogeneous parameter is set to 0, instead, the genotype includes only N genes which are used to generate identical networks. The step function returns the reward for the predator robot. The reward is always 0 unless in the last step in which it is 1.0 minus the fraction of steps required to catch the prey (0.0 if the predator did not catch the prey). The reward for the prey corresponds to the inverse of that of the predator (i.e. 1.0 minus the reward of the predator).

The source code (predprey.cpp, predprey.h, utilities.cpp, utilities.h, robot-env.cpp, robot-env.h, ErPredprey.pxd, ErPredprey.pyx, and setupErDiscrim.py) can be compiled from the evorobotpy/lib folder with the following command:

```
python3 setupErPredprey.py build_ext --inplace
cp ErPredprey*.so ../bin # or cp ErPredprey*.dll ../bin
```

The co-evolution of competitive robots requires a special algorithm that operates with two populations and discriminate global progress, i.e. progress with respect to any kind of competitor from local progress, i.e. progress against current competitors which however lead to retrogression against different competitors (Simione and Nolfi, 2019). Such algorithm is implemented in the /bin/coevo2.py script and can be selected by specifying *algo* = *coevo2* in the [ADAPT] section of the .ini configuration file.

The ErPredprey environment uses a special syntax to indicate the robots that you want to post-evaluate with the *–t* parameter. In particular, you can use the string "-t P-1000-5" as in the command shown below to show the behavior of the best 5 predator robots of generation 1000 against the best 5 prey robots of generation 1000.

```
python3 ../bin/es.py –f ERPredprey.ini –s 12 –t P–1000–5
```

Moreover, you can use the string "-t m-1000-100" to post-evaluate the predator and prey of generation 1000 against predator and prey of previous generations, every 100 generations.

4. Reinforcement Learning

Reinforcement learning algorithm are more complex than evolutionary algorithm and consequently more difficult to implement. The reader can however rely on library such as Spinningup and Baselines to familiarize with high quality implementation of reinforcement learning algorithms and to run experiments with these methods. Spinningup is ideal to start familiarizing with these methods since it includes a gentle introduction to the area, compact and well-documented source code, exercises and pointers to the relevant literature. Baselines includes highly quality implementations of reinforcement learning algorithms. Both tools require Tensorflow, (https://github.com/tensorflow/tensorflow),a library for large-scale machine learning (Abadi et al., 2016).

4.1 Spinningup

Spinningup (https://spinningup.openai.com/) is a free educational resource developed by Josh Achiam that enables the acquisition of theoretical and practical knowledge on reinforcement learning integrated with openAI-Gym. You can refer to the documentation included in the website indicated above for installation and usage.

4.2 Baselines

Baselines (https://github.com/openai/baselines) is free library developed by OpenAI that contains carefully designed implementation of several state-of-the-art reinforcement learning algorithm (e.g. PPO, TRPO, A2C, ext.) interfaced with OpenAI gym.

Training can be launched with the following command from the /baselines folder:

```
python -m baselines.run --alg=<name of the algorithm> --env=<environment_id> [additional arguments]
```

For example:

```
OPENAI_LOGDIR=./logs/Humanoid/ OPENAI_LOG_FORMAT=csv python -m baselines.run --alg=ppo2 --env=HumanoidBulletEnv-v0 --network=mlp --num_hidden=256 --num_layers=2 --num_timesteps=5e7 --seed=1 --save_path=./models/humanoid --save_interval=500
```

will train a policy for the HumanoidBullet environment for 5 * 10^7 steps by using a fully connected feed-forward network with 2 internal layers each including 256 neurons. The random seed is set to 1. Statistics will be saved in CSV file named progress.csv located in the ./logs/Humanoid/ folder. The trained policy will be saved in a the file ./models/humanoid. The state of the training network is saved every 500 epochs in files located in the ./log/Humanoid/checkpoints/ folder. Epochs last 2048 steps by default.

To observe the behavior of the trained policy you should specify the --play parameter, set *num_timesteps* to 0, and indicate with the *–load_path* parameter the file to be loaded. Notice, however, that for PyBullet environments you need to add the instruction env.render(mode="human") in the method train() of the file ./baselines/run.py. Otherwise, bullet will not display the robot's behavior. If you want to load the policy at an intermediate state of training, you should specify the name of one of the files saved in the ./log/Humanoid/checkpoints/ folder. For example, to inspect the behavior produced by the robot trained with the command described at the beginning of this section, you can use the following command:

```
python -m baselines.run --alg=ppo2 --env=HumanoidBulletEnv-v0 --num_hidden=256 --num_layers=2 --num_timesteps=0 --seed=1  --load_path=./models/humanoid --num_env=1 --play
```

In summary, the basic parameters that you might want to use are the following:

```
OPENAI_LOGDIR=./folder                  directory used to save files
OPENAI_LOG_FORMAT=csv                   save statistics in a csv file
--alg=<algoname>                        name of the algorithm used
--num_layers=N                          number of layers of the policy network
--num_hidden=N                          number of hiddens of the policy network
--num_timesteps=2e7                     total number of training timesteps
--seed=1                                seed number initializator
--save_path=<file-path>                       filename used to save the trained policy
--load_path=<file-path>                 filename containing the trained policy
--save_interval=N                       the frequency with which the model is saved
--play                                  graphically render the behavior (set num_timesteps=0)
```

For a description of additional parameters see the source code. For example, for a description of the parameters of the PPO2 algorithm you can check the file ppo2.py.

Finally, you can plot the average return data contained in the .csv file with the visualization tool provided by baselines and described in: https://github.com/openai/baselines/blob/master/docs/viz/viz.ipynb or by implementing a custom script for plotting the data included in the .csv file.

Exercise 7. Train the policy for a pybullet locomotor problem with the PPO algorithm (do not forget to restore the original pybullet files with the reward functions suitable for reinforcement learning, as described in Section 3.3). Compare the result with those obtained with evolutionary strategies.

Acknowledgements

The author thanks Paolo Pagliuca who contributed significantly to the implementation evorobotpy tool and Vladislav Kurenkov who prepared the docker container and provided useful feedbacks on the first draft of this paper.

References

Abadi M., Barham P., Chen J., Chen Z., Davis A., Dean J., ... & Kudlur M. (2016). Tensorflow: A system for large-scale machine learning. In 12th Symposium on Operating Systems Design and Implementation (16): 265-283.

Bonani, M., Longchamp, V., Magnenat, S., Rétornaz, P., Burnier, D., Roulet, G., Vaussard, F., Bleuler, H., & Mondada, F. (2010). The marXbot, a miniature mobile robot opening new perspectives for the collective robotic research. In Proceedings of the 2010 IEEE/RSJ international conference on intelligent robots and systems (IROS 2010) (pp. 4187–4193). New York: IEEE Press.

Brockman G., Cheung V., Pettersson L, Schneider J., Schulman J., Tang J. and Zaremba W. (2016). OpenAI Gym. arXiv:1606.01540

Coumans E. & Yunfei Bai (2016-2019). PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org, https://github.com/erwincoumans/pybullet_robots, https://github.com/robotology-playground/pybullet-robot-envs

Lehman J. & Stanley K.O. (2011). Abandoning objectives: Evolution through the search for novelty alone. Evolutionary Computation, 19 (2): 189-223.

Massera G., Ferrauto T., Gigliotta O., and Nolfi S. (2013). FARSA: An open software tool for embodied cognitive science. In P. Lio', O. Miglino, G. Nicosia, S. Nolfi and M. Pavone (Eds.), Proceeding of the 12th European Conference on Artificial Life. Cambridge, MA: MIT Press.

Massera G., Ferrauto T., Gigliotta O., Nolfi S. (2014). Designing adaptive humanoid robots through the FARSA open-source framework. Adaptive Behavior, 22 (3):255-265

Mondada F., Bonani M., Raemy X. et al. (2009). The e-puck, a robot designed for education in engineering. Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions.

Mondada, R., Franzi, E. & Ienne, P. (1993) Mobile robot miniaturization: A tool for investigation in control algorithms. In: Proceedings of the Third International Symposium on Experimental Robots, eds. T. Yoshikawa & F. Miyazaki. Kyoto, Japan.

Nolfi S. (2000) Evorobot 1.1 user manual. Technical Report, Roma, Italy: Institute of Psychology, National Research Council.

Nolfi S. and Gigliotta O. (2010). Evorobot*: A tool for running experiments on the evolution of communication. In S. Nolfi & M. Mirolli (Eds.), Evolution of Communication and Language in Embodied Agents. Berlin: Springer Verlag.

Pagliuca P., Milano N. & Nolfi S. (2018). Maximizing adaptive power in neuroevolution. PLoS One, 13(7): e0198788.

Salimans T., Ho J., Chen X., Sidor S & Sutskever I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. arXiv:1703.03864v2

Sandini G., Metta G. & Vernon D. (2004). Robotcub: An open framework for research in embodied cognition, International Journal of Humanoid Robotics (8) 2: 18–31.

Simione L. & Nolfi S. (2019). Long-term progress and behavior complexification in competitive co-evolution. arXiv:1909.08303.