Using OpenAI with ROS

09/02/2018

OpenAI has released the
Gym, a toolkit for developing and comparing reinforcement learning (RL) algorithms. That toolkit is a huge opportunity for speeding up the progress in the creation of better reinforcement algorithms, since it

provides an easy way of comparing them, on the same conditions, independently of where the algorithm is executed.

The toolkit is mainly aimed at the creation of RL algorithms for a general abstract agent. Here, we are interested in applying it to the control of robots (of course!). Specifically, we are interested in ROS based robots. That is why, in this post we describe how to apply the OpenAI Gym to the control of a drone that runs with ROS

Let's see an example of training.

The drone training example

In this example, we are going to train a ROS based drone to be able to go to a moving as low as possible (may be to avoid being detected), but avoiding obstacles in its way.

For developing the algorithm we are going to use the ROS Development Studi (RDS). That is an environment that allows to program with ROS and its simulations with a web browser, without having to install anything on the computer. So we have all the required packag and Gazebo simulations already installed. You can follow the rest of the post you have two options:

1. Either go to the **ROS Development Studio** and create a free account.
2. Either you install everything in your computer. If you need to install those packa **here** for ROS installation, **here** for OpenAI installation, and **here** to download the Ga (for Indigo + Gazebo 7).

What follows are the instructions for developing the training program with RDS, but as you can see, the steps are the same for your local installation.

Get the prepared code for drone training

We have prepared the training code already for you, so you don't have to build
The goal of this post is to show you how this
code works, and how you would modify it for your own case (different robot, or different task).

In order to get the code, just open the RDS (http://rds.theconstructsim.com) and create a new project. You can call it *openai_with_ros_example*. Then open the project by clicking on the *Open Project* button.

Once you have the environment open, go to the *Tools* menu and open a Linux *Shell*. Inside the shell go to the *catkin_ws/src* directory. This is the place where ROS code must be put in the RDS in order to build, test, debug and execute it against robot simulations. Once there, clone the following git repo which contains the code to train the drone with OpenAI:

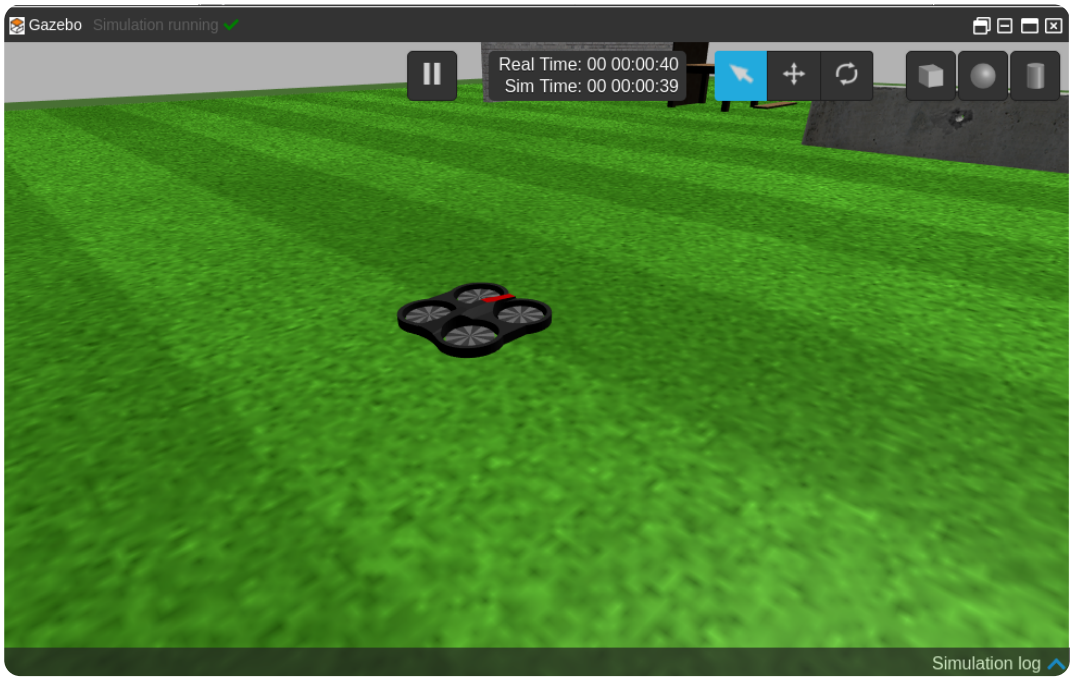
```
git clone https://bitbucket.org/theconstructcore/drone_trai
```

Now, you should have a ROS package named *drone_training* inside the *catkin_ws*. **Let's test it right now!**

Testing what you have installed

First thing would be to test what you got so we can see what we are trying to understand. For this, follow the next steps:

1. Launch the Parrot drone simulation. On RDS, you can find it as *Parrot AR.Drone* at the menu *Simulations*. After launching, you show see a window like this.

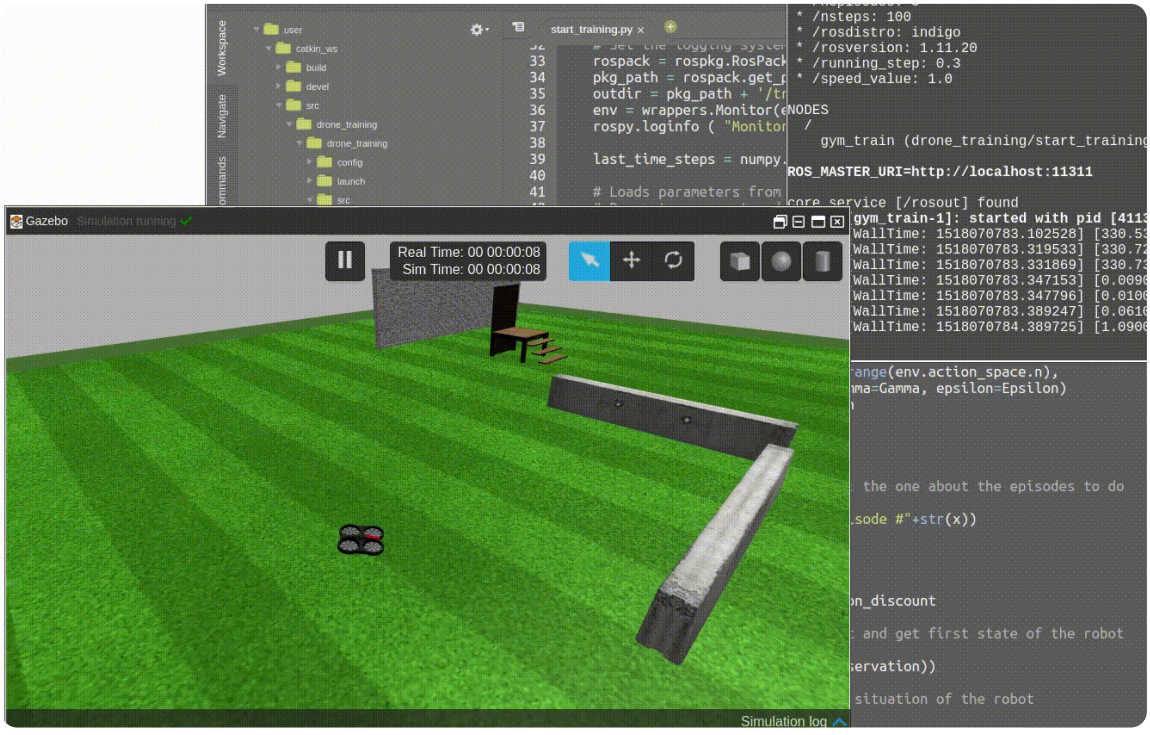


Parrot Drone with ROS simulation

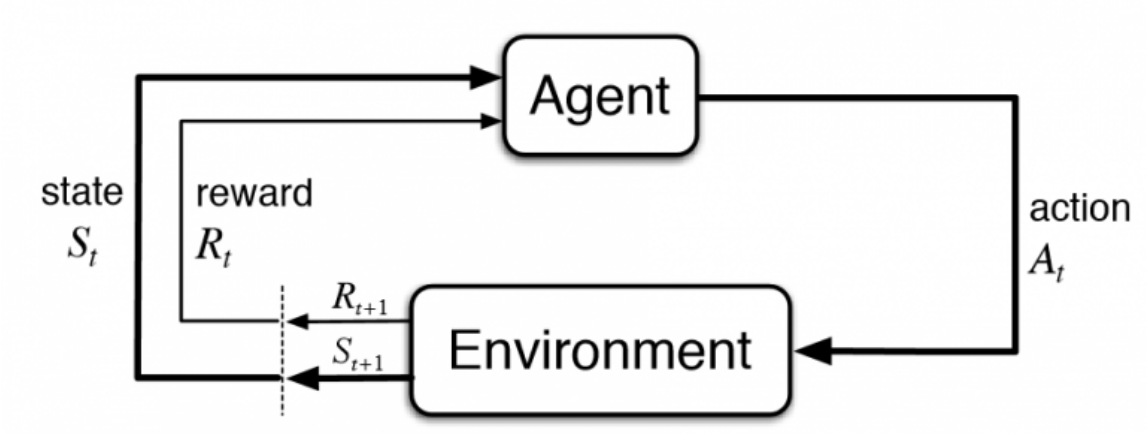
1. Let's launch our package so it will start training the Parrot drone. For that, type the following in the previous shell:

```
> roslaunch drone_training main.launch
```

You should see the drone start moving doing some strange things. It actually looks like the drone is drunk! That makes perfect sense.



What is happening is that the robot is learning. It is exploring its space of actions and practicing what it will sense based on the actions that it takes. That is exactly how the reinforcement learning problem works. Basically, the robot is performing the classical RL loop o the figure:



How a reinforcement learning problem works (image from StackOverflow)

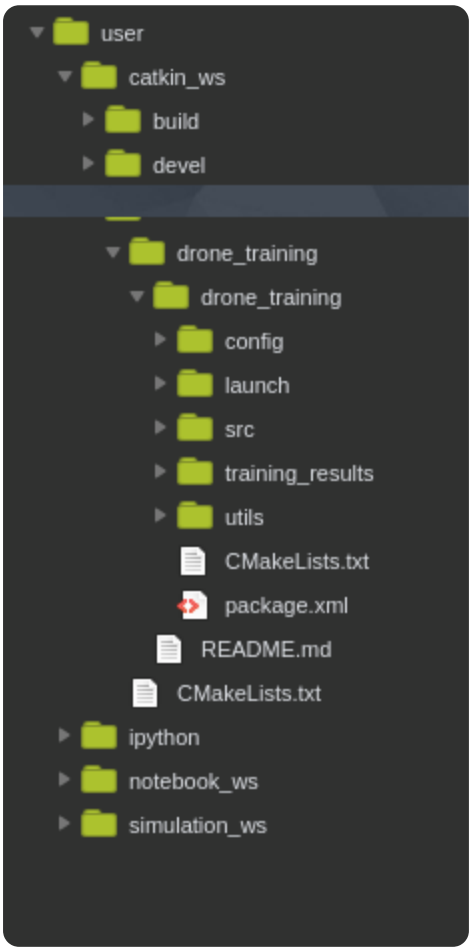
The **agent** (the drone plus the learning algorithm), decides to take an **action** from the pool of available actions (for example, move forward), and executes it in the **environment** (the drone moves forward). The result of that action, makes the agent closer or not to its target (to fly to a given location). If the robot is closer, it gets a good **reward.** If it is further away, it gets a bad reward.
In any case,
the agent perceives the current **state** of itself and the environment (where it is located now), and then feeds reward,
previous state, new state and action taken to the learning algorithm (to learn the results based on its actions). Then the process repeats again for the **number of steps** the robot is allowed to experiment. When the number of steps is done, the final reward is obtained and the robot starts again from the initial position, now with an improved algorithm. The whole process is repeated again and again for a given **number of episodes** (usually high).

Now, let's see how all that works together in the code. Let's see its structure:

The *drone_training* package for OpenAI training with ROS

This package is just an example of how you can interface OpenAI with ROS robots. There are other ways of doing it, and in future posts we will explore them.

The package contains the following directories:



- *launch* directory. It contains the *main.launch* file that we used to launch the whole training thing.
- *config* directory. It contains a configuration file *qlearn_params.yaml* with the desired parameters for the training. Usually, you need to tune the parameters by trying several times, so it is a good practice to keep them structured on a file or files for easier modification/review.
- *training_results* directory. It will contain the results of our training for later analysis.
- *utils* directory. Contains a Python file *plot_results.py* that we will use to plot the training results.
- *src* directory. The crux of the matter. It contains the code that makes possible the training of the drone. Let's have a deeper look at this one.

The src directory

The launch file will launch the *start_training.py* file. That is the file that orchestrates the training. Let's see what it does step by step:

Initializes the ROS node

```
rospy.init_node('drone_gym', anonymous=True)
```

Of course, first thing is to declare that code as a node of ROS.

Creates the Gym environment

```
env = gym.make('QuadcopterLiveShow-v0')
```

That is the main class that OpenAI provides. **Every experiment of OpenAI must be defined within an environment**. By organizing like that, different developers can test different algorithms always comparing against the same environment. Hence we can compare if an algorithm is better than another always on the same conditions.

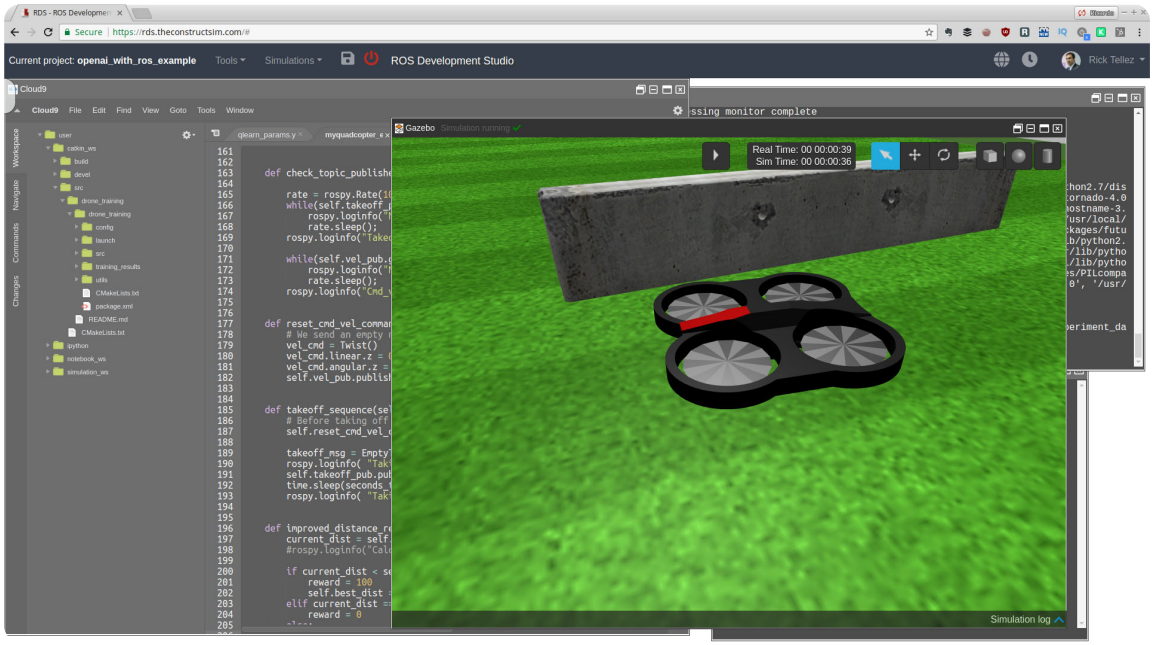
The environment defines the actions available to the agent, how to compute the reward based on its actions and results, and how to obtain the state of the world of the agent.

Every environment in OpenAI must define the following things:

- A function *_reset* that sets the training environment to its initial state.
- A function *_step* that makes the changes in the environment based on the last action taken, and then observes what the new state of the environment is. Based on those two things, it generates a reward.
- A function *_seed* used for initializing the random number generator.
- A function *render* function used to show on screen what is happening in the environment.
- What is the task that the agent has to solve in this environment.
- The number of possible actions in the environment. In our case, we are allowing the robot to take 5 actions.
- A way to compute the reward obtained by the agent. Those are the points provided to the agent on each step, based on how good or how bad has it done in the environment to solve the task at hands.

- A way to determine if the task at hands has been solved.

We are going to see how to do the code of the environment below.



Loading the parameters of the algorithm from the ROS param server

```
Alpha = rospy.get_param("/alpha")
Epsilon = rospy.get_param("/epsilon")
Gamma = rospy.get_param("/gamma")
epsilon_discount = rospy.get_param("/epsilon_discount")
nepisodes = rospy.get_param("/nepisodes")
nsteps = rospy.get_param("/nsteps")
```

Those are the parameters that our learning algorithm needs. That section will change based on the parameters your algorithm needs.

Initialization of the learning algorithm

We create an instance of the learning algorithm we are going to use.

```
qlearn = qlearn.QLearn(actions=range(env.action_space.n), a
```

In this case we are using a Qlearning reinforcement learning algorithm. But yo
available (including deep learning) or encode your own.

That is the key part that we want to test. How good is this algorithm for solving the task at hands.

Implementation of the training loop

The training loop is the one that repeats the learning cycle explained above. That is where the learning code is executed. It basically consists of two main loops:

- First we have a loop with the **number of episodes** that the robot will be tested. Each episode means the number of times that we will allow the robot to try to solve the task.
- Second, we have the loop with the **number of steps**. For each episode, we allow the robot to take a given number of actions, number of steps, number of loops into the reinforcement cycle. If the robot consumes all the steps, we consider it has not solved the task and hence, a new **episode** must start.

The number of episodes loop

It starts with the code:

```
for x in range(nepisodes):
```

Remember that the number of episodes is a parameter from the config file. Th
to reset the environment (initialize the robot) so a new trial can start from the original position. It also gets the initial state observation required by the learning algorithm to generate the first action.

```
observation = env.reset()
```

The number of steps loop

It starts with the code:

```
for i in range(nsteps):
```

and basically what it does is:

- Make the learning algorithm choose an action based on the current state

```
action = qlearn.chooseAction(state)
```

- Execute the action in the environment

```
observation, reward, done, info = env.step(action)
```

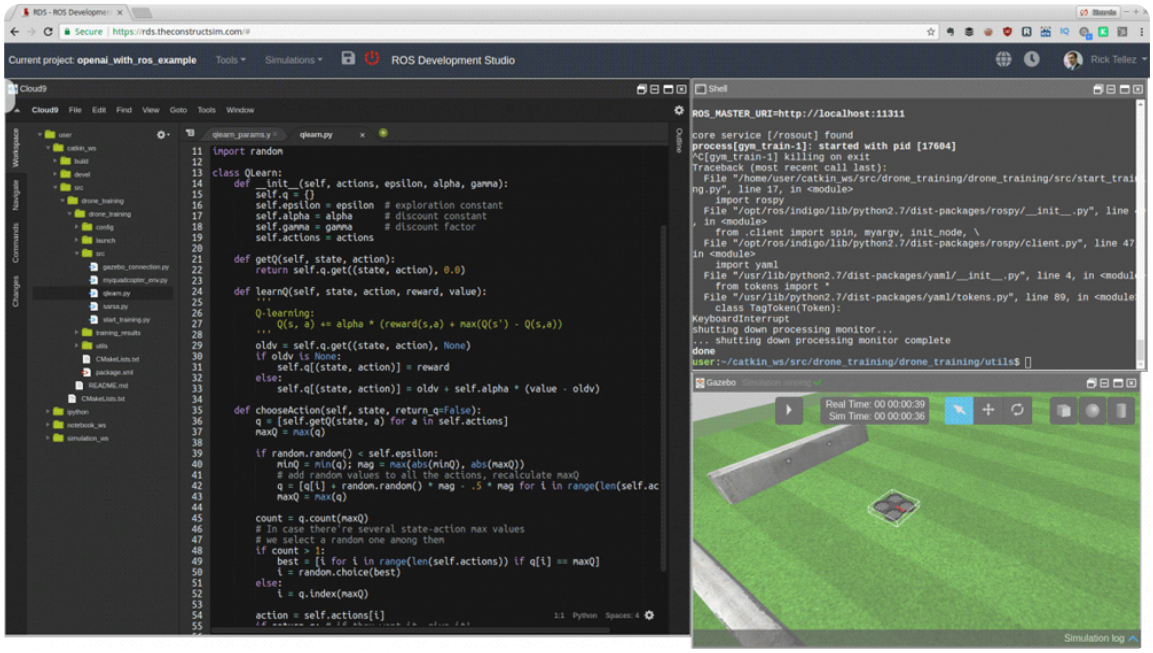
- Get the new state after the action

```
nextState = ''.join(map(str, observation))
```

- Learn from that result

```
qlearn.learn(state, action, reward, nextState)
```

And that is all. That simple. The loops will repeat based on the parameters values, and once they finish, the log files will generate in the *training_results* directory.



About the learning algorithm

In this example, we are using the Qlearn reinforcement learning algorithm. That is a classical algorithm of reinforcement learning. You can **find here a description of it**.

The code for the Qlearn algorithm is provided in the *qlearn.py* file. It has been taken from **Victor Mayoral's git**, and you can find the original code h (thanks Victor for such a good work!).

You could change this algorithm for another one that you may have developed next hit in artificial intelligence. Just create the code (like the *qlearn.py*) with the same inputs and outputs, and then substitute the call in the *start_training.py* file. That is the greatness of the OpenAI framework: that you can just plug your algorithr anything of the rest, and the whole learning system will still work. By doing this, you can compare your algorithm with the others under the exact same conditions.

Additionally, we have included in the repo another classic reinforcement learni (*sarsa.py*).

Here your first homework!

Change the learning algorithm inside the *start_training.py* file by the Sarsa algorithm, and watch if there is any difference in learning speed or improved behavior.

The Gym environment

As I said, the environment defines the actions available to the agent, how to compute the reward based on its actions and results, and how to obtain the state of the world of the agent, after that actions have been performed.

OpenAI provides an standarized way of creating an environment. Basically, yo environment class which must inherit from *gym.Env*. That inheritance, entitles you to implement within that cla functions *_seed, _reset* and *_step* (explained above).

In our case, we have created a class named *QuadCopterEnv*. You can find the code in the *myquadcopter_env.py* file.

The code starts by registering the class into the pool of available environments of OpenAI. You register a new environment with the following code:

```
reg = register(
 id='QuadcopterLiveShow-v0',
 entry_point='myquadcopter_env:QuadCopterEnv',
 timestep_limit=100,
 )
```

Then the class starts initializing the topics it needs to connect to, gets the configuration parameters from the ROS param server, and connects to the Gazebo simulation.

```
 def __init__(self):

   self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_
   self.takeoff_pub = rospy.Publisher('/drone/takeoff', Emp

   self.speed_value = rospy.get_param("/speed_value")
   self.desired_pose = Pose()
   self.desired_pose.position.z = rospy.get_param("/desirec
   self.desired_pose.position.x = rospy.get_param("/desirec
   self.desired_pose.position.y = rospy.get_param("/desirec
   self.running_step = rospy.get_param("/running_step")
   self.max_incl = rospy.get_param("/max_incl")
   self.max_altitude = rospy.get_param("/max_altitude")

   self.gazebo = GazeboConnection()

   self.action_space = spaces.Discrete(5) #Forward,Left,Rig
   self.reward_range = (-np.inf, np.inf)
```

```
        self._seed()
```

Now it is time for the definition of each of the mandatory functions for an environment.

- In the function *_seed*, we initialize the random seed, required to generate random numbers. Those are used by the learning algorithm when generating random actions
- In the function *_reset* we initialize the whole environment to a known initial state, so all the episodes can start with the same conditions. Very simple code. Just resets the simulation, and clears the topics. The only special thing is that at the end of the function we pause the simulation, because we do not want the robot be running while we are doing other computational tasks. Otherwise, we would not be able to guarantee the initial conditions for all the episodes, since it would largely depend on the execution time of other algorithms in the training computer.
- The main function is the *_step* function. This is the one that is called during the loops of training. This function receives as a parameter the action selected by the learning algorithm. Remember that that parameter is just the number of the action select, not the actual action. The learning algorithm doesn't know which actions we have for this task. It just knows the number of actions available and it picks one of them based on its current learning status. So what we receive here is just the number of the action selected by the learning algorithm.
  - The first thing we do is to convert that number into the actual action command for the robot. That is what this code does, it converts the number into the movement action that we will send to the ROS robot:

```
        vel_cmd = Twist()
        if action == 0: #FORWARD
        vel_cmd.linear.x = self.speed_value
        vel_cmd.angular.z = 0.0
```

```
elif action == 1: #LEFT
vel_cmd.linear.x = 0.05
vel_cmd.angular.z = self.speed_value
elif action == 2: #RIGHT
vel_cmd.linear.x = 0.05
vel_cmd.angular.z = -self.speed_value
elif action == 3: #Up
vel_cmd.linear.z = self.speed_value
vel_cmd.angular.z = 0.0
elif action == 4: #Down
vel_cmd.linear.z = -self.speed_value
vel_cmd.angular.z = 0.0
```

- Next step is to send the action to the robot. For that, we need to unpause the simulator, send the command, wait for some time for the execution of the command, take an observation of the state of the environment after the execution, and pause again the simulator.

```
self.gazebo.unpauseSim()
self.vel_pub.publish(vel_cmd)
time.sleep(self.running_step)
data_pose, data_imu = self.take_observation()
self.gazebo.pauseSim()
```

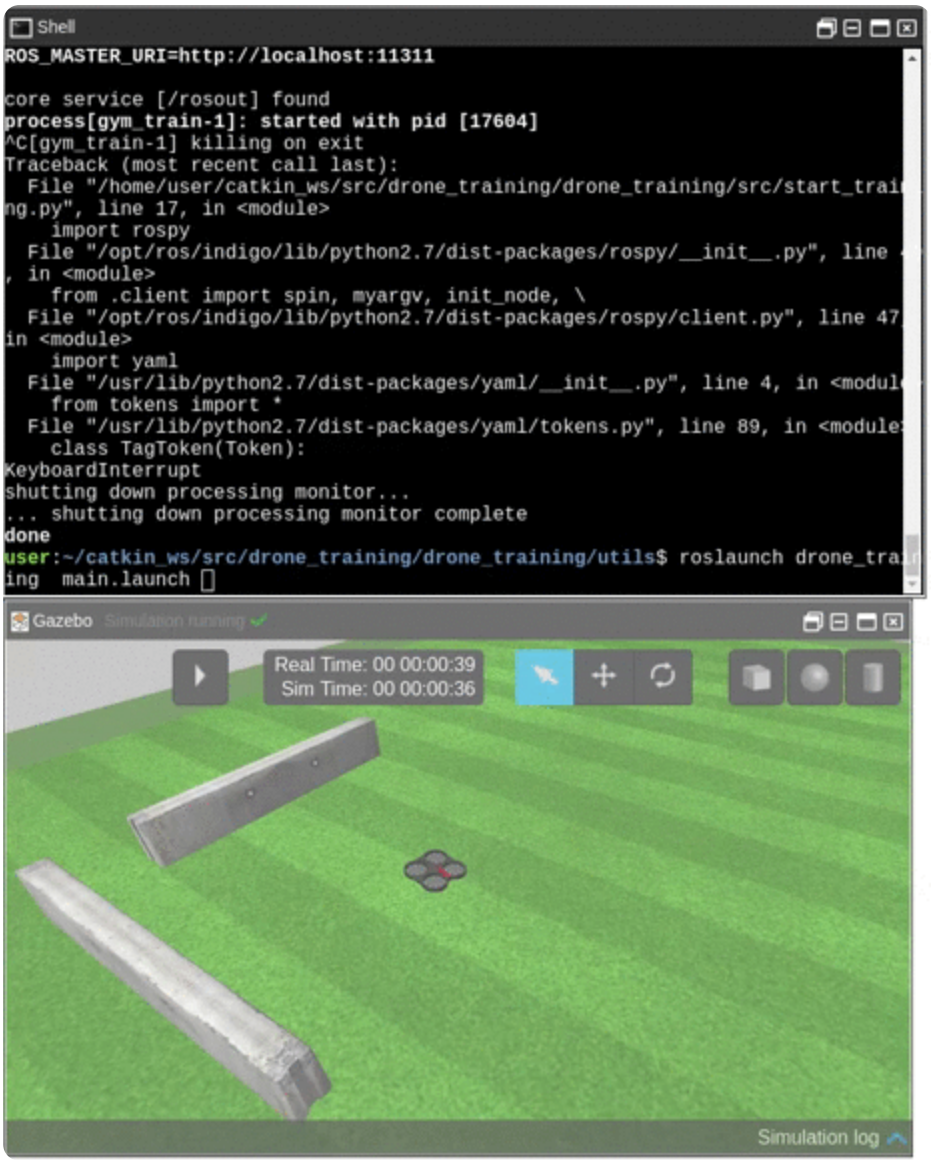
- Then, we process the current state of the robot/environment to calculate the reward. For the reward, we are taking into account how close to the desired position the drone is, but also other factors like the inclination of the drone, or it height. Additionally, we promote moving forward against turning.

```
reward,done = self.process_data(data_pose, data_ir
if action == 0:
  reward += 100
elif action == 1 or action == 2:
  reward -= 50
```

```
        elif action == 3:
            reward -= 150
        else:
            reward -= 50
```

- Finally, we return the current state, the reward obtained and a flag indicating be considered done (either because the drone achieved the goal, or either because it went against the conditions of height or inclination).

```
        state = [data_pose.position.x]
        return state, reward, done, {}
```



Basically, that's it. The code above calls additional functions that you can check by looking into the QuadCopterEnv class. Those are the functions that do the dirty job of calculating the actual values, but we don't need to review them here, because they are out of the scope of this subject.

One function that we may need to cover, though, is the function that computes the reward. Its code is the following:

```
def process_data(self, data_position, data_imu):
  done = False
  euler = tf.transformations.euler_from_quaternion([data_imu
                                                    data_imu
                                                    data_imu
                                                    data_imu

  roll = euler[0]
  pitch = euler[1]
  yaw = euler[2]

  pitch_bad = not(-self.max_incl < pitch < self.max_incl)
  roll_bad = not(-self.max_incl < roll < self.max_incl)
  altitude_bad = data_position.position.z > self.max_altituc

  if altitude_bad or pitch_bad or roll_bad:
    rospy.loginfo ("(Drone flight status is wrong) >>> ("+st
    done = True
    reward = -200
  else:
    reward = self.improved_distance_reward(data_position)

  return reward,done
```

That code, basically does two things:

1. First, detects if the robot has surpassed the previously defined operation limits of height and inclination. If that is the case, it considers the episode done
2. Computes the reward based on the distance to the goal

How to configure the test

You can find a *yaml* file in the *config* directory containing the different parameters required to configure the learning task. I have divided the parameters in two types:

1. Parameters related to the learning algorithm being used: those are the parameters that my algorithm needs. In this case, are specifically for a Qlearn algorithm. You would define here the ones your algorithm needs, and then read them in the *start_training.py* file
2. Parameters related to the environment: those are parameters that affect the wa obtained, and hence, they affect the environment. Those include the goal position or the conditions f can be considered aborted due to unstable drone conditions (too much altitude or too much inclination).

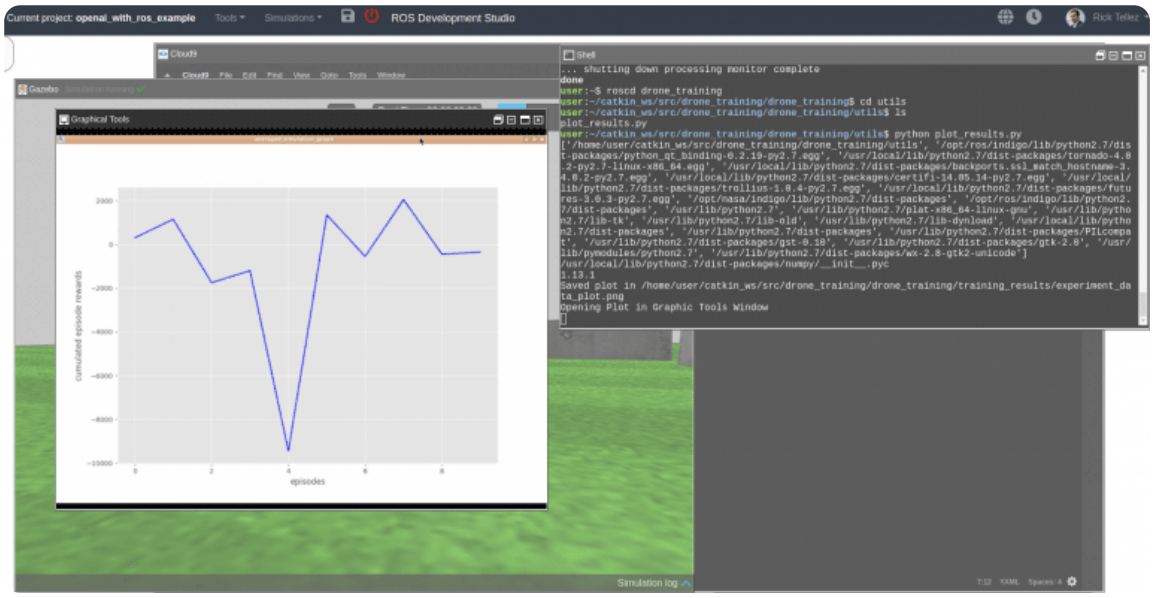
How to plot the results of the training

Plotting results is very important because you can visually identify if your syste and how fast. If you are able to early identify that your system is not learning properly, then you can modify the parameters (or even the conditions of the experiment), so you can retry fast.

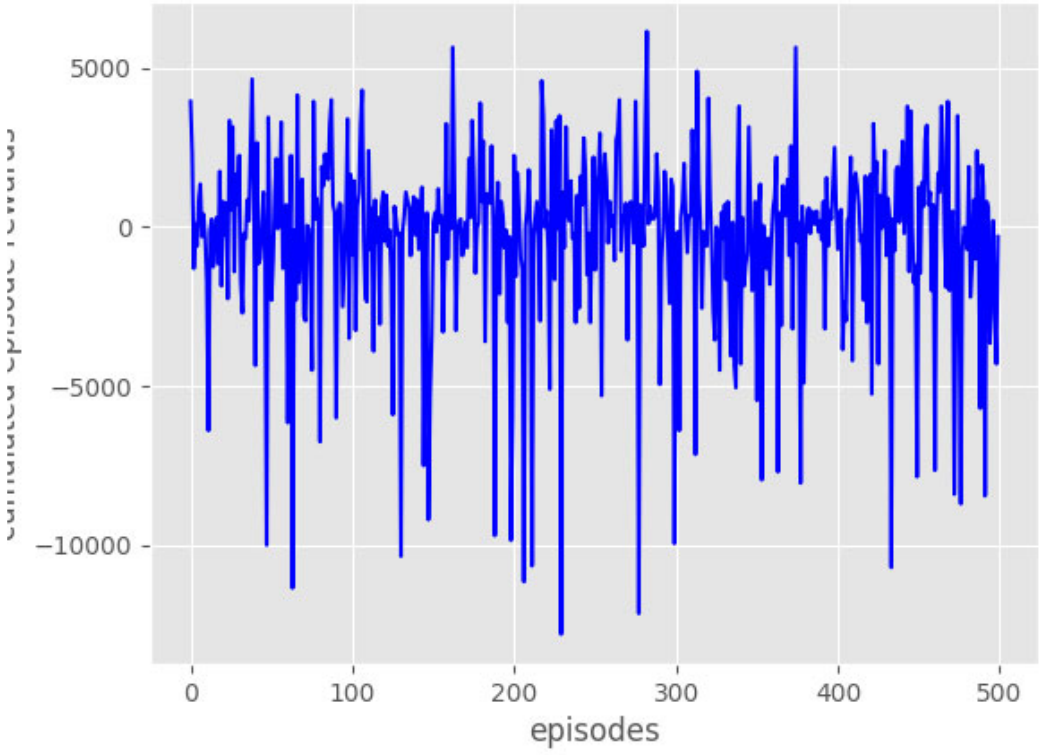
In order to plot the results, I have provided a Python script in the *utils* directory that does the job. I did not create the code myself, I took it from somewhere else, but I cannot remember from w (if you are the author and want to have the credit just contact me). To launch that code just type on the *utils* directory:

```
> python plot_results.py
```

The script will take the results generated in the training_results directory and generate a plot with all the rewards obtained for each episode. In order to watch the plot, you must open the *Graphic Tools* window (*Tools->Graphic Tools*). You must see something like this:



For this post, I have run the code provided to you (as the version of 8 Feb2018) for 500 episodes, and the results are not very good, as you can see in the following figure



In that figure, you can see that there is no progress in the reward, episode after episode. Furthermore, the variations in the reward values look completely random. This means that the algorithm is actually not learning at all about the problem trying to solve. What can be the reasons for that? Well, I can figure out a few. The goal for the engineer is to devise ways to modify the learning situation so the learning can actually be accomplished. Some possible reasons why is not learning:

- The reward function is not properly created. If the reward is too complex, the system may not be able to capture small baby steps of improvement.
- The state provided to the learning algorithm is continuous, and Qlearning is r that. As you can see in the code, we are returning the current robot position in the x axis as the state That is a continuous value. I would suggest to discretize the state into different zones (let's say 10 zones). Each zone meaning getting closer and closer to the goal point.
- The parameters of the learning algorithm are not correct.
- The experiment is not completely sound *per se*. Take into account that the goal position of the robot is fixed, and that the obstacles cannot be detected by the robot (he can only infer their position after crashing many times against it in several episodes).

What is clear is that the structure of the training environment is correct (by structure, I mean the organization of the whole learning system). That is a good point, since it allow us to start looking for ways to improve the learning from within a learning structure that already works.

How to improve the learning

This example is massively improvable (as the plot of the results are showing

;-). Here a list of suggestions where you can improve the system to make it lea solutions:

- Take the observations/state in (x,y, z) instead of only x. Also, discretize the state space.
- Make the robot detect obstacles with sonar sensor and use it to avoid obstacles.
- Make the robot go to random points, not only to a fixed point

That is your homework!

Apply any of those improvements, and send me your plots of the improved reward evolution, and videos showing how the drone has learnt to do the task. We will publish them in our social channels giving you credit about it.

ROS Developers Live Show about this example OpenAI with ROS

We recently did a live class showing how all the explained above works in real time with many people attending at the same time and doing the exercises with me. It may clarify you all the content above. Have a look here:

We do a ROS Developers Live Show every Wednesday at 18:00 CET. You may want to **subscribe** to **our Youtube channel** in order to

stay notified of our future Live Show.

Additionally, we have created an online course where to learn all the material a about OpenAI for robotics.
It is online with all the simulations integrated and requires just a web browser (you can do the exercises with ROS even from Windows!). You can find it here: **OpenAI Gym for Robotics 101** (additionally, in case you like it, you can use the discount coupon 2AACEE38 for a 10% discount).

Conclusion

OpenAI is a very good framework for training robots to do things using the latest techniques in artificial intelligence. Also, as you have seen, it is not difficult to integrate with ROS based robots. This makes the tandem OpenAI+ROS a killer combination for robot development!

If you still have doubts, write your questions below the post and we will try to answer them all. Happy robot training!

Topics: OpenAI Gym | ROS

Check Out These Related Posts

Teaching Robotics to University Students from Home

Jul 15, 2020

The world has changed in 2020. Due to the coronavirus, all our social interactions have been...

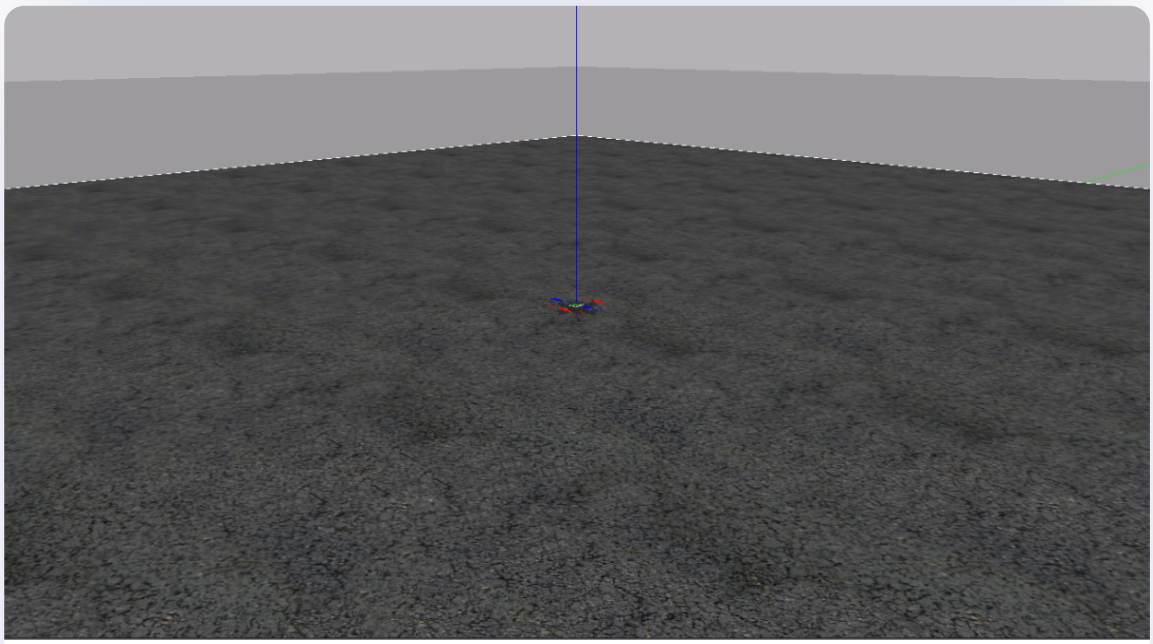
read more

[ROS2 How-to] #2 – Create a ROS2 action server

May 20, 2022

What we are going to learn - How to create a custom action message How to create an action server...

read more



Program Drones using ROS2 – Episode 1

May 18, 2022

In this post, we will see how to program drones using ROS2. Perhaps you have programmed drones...

read more

« Older Entries

0 Comments

Trackbacks/Pingbacks

1. [RDS] 004 - ROS Development Studio #Howto use git in RDS | The Construct - [...] 4. If you want more details about the drone_training package you can go to https://www.theconstructsim.com/using-openai-ros/ 5. The simulation was...

SUPPORT

Contact Us

Forum

The Construct Help Center

T (+34) 687 672 123

info@theconstructsim.com

Privacy Policy

Terms of Use