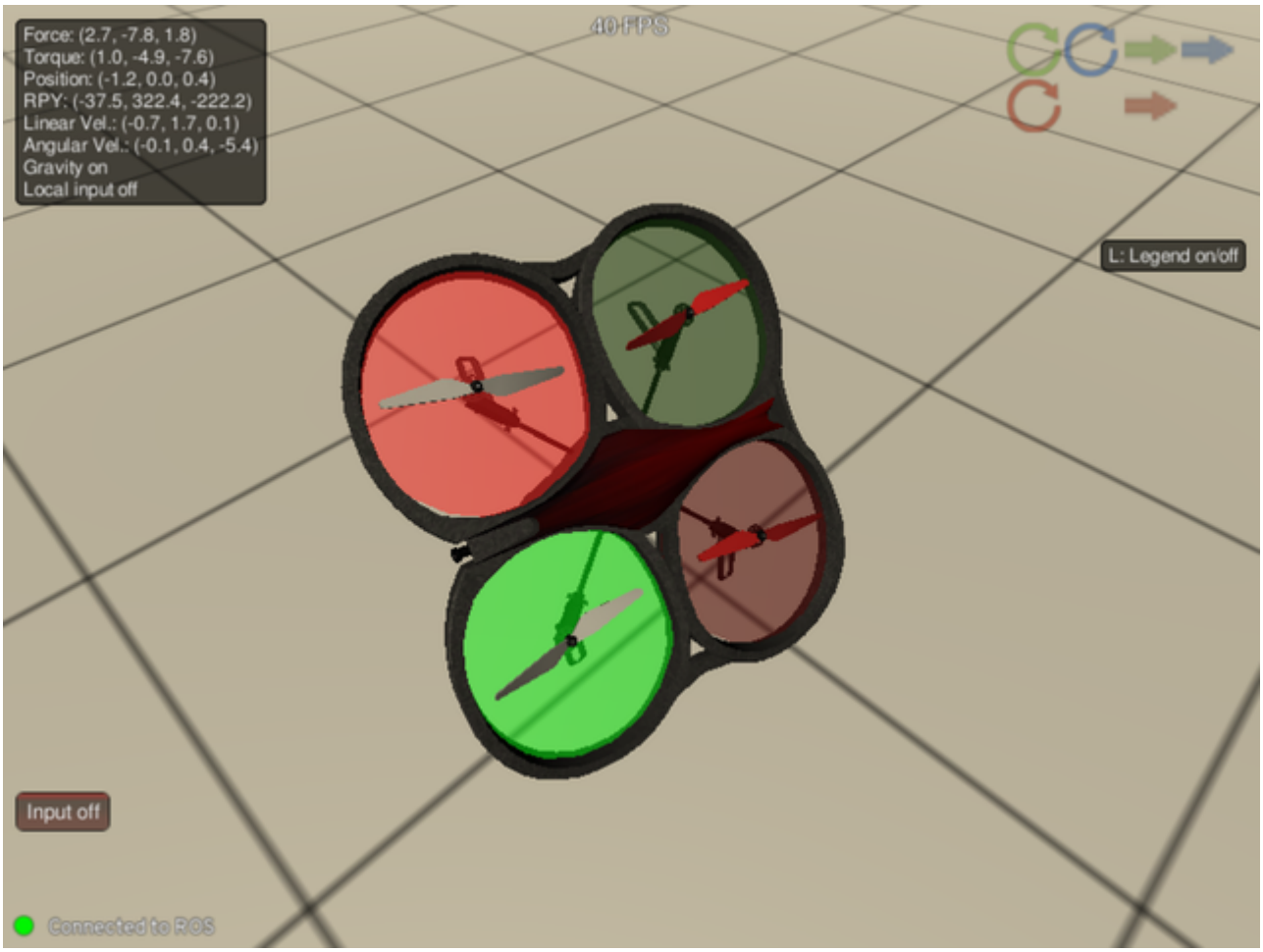
Project: Train a Quadcopter How to Fly

Design an agent that can fly a quadcopter, and then train it using a reinforcement learning algorithm of your choice! Try to apply the techniques you have learnt, but also feel free to come up with innovative ideas and test them.



Instructions

> **Note**: If you haven't done so already, follow the steps in this repo's README to install ROS, and ensure that the simulator is running and correctly connecting to ROS.

When you are ready to start coding, take a look at the `quad_controller_rl/src/` (source) directory to better understand the structure. Here are some of the salient items:

- `src/`: Contains all the source code for the project.
  - `quad_controller_rl/`: This is the root of the Python package you'll be working in.
  - ...
  - `tasks/`: Define your tasks (environments) in this sub-directory.
    - `__init__.py`: When you define a new task, you'll have to import it here.
    - `base_task.py`: Generic base class for all tasks, with documentation.
    - `takeoff.py`: This is the first task, already defined for you, and set to run by default.
  - ...
  - `agents/`: Develop your reinforcement learning agents here.

    - `__init__.py`: When you define a new agent, you'll have to import it here, just like tasks.
    - `base_agent.py`: Generic base class for all agents, with documentation.
    - `policy_search.py`: A sample agent has been provided here, and is set to run by default.
- ...

Tasks

Open up the base class for tasks, `BaseTask`, defined in `tasks/base_task.py`:

```
class BaseTask:
    """Generic base class for reinforcement learning tasks."""

    def __init__(self):
        """Define state and action spaces, initialize other task parameter
s."""
        pass

    def set_agent(self, agent):
        """Set an agent to carry out this task; to be called from updat
e."""
        self.agent = agent

    def reset(self):
        """Reset task and return initial condition."""
        raise NotImplementedError

    def update(self, timestamp, pose, angular_velocity, linear_acceleratio
n):
        """Process current data, call agent, return action and done fla
g."""
        raise NotImplementedError
```

All tasks must inherit from this class to function properly. You will need to override the `reset()` and `update()` methods when defining a task, otherwise you will get `NotImplementedError`'s. Besides these two, you should define the state (observation) space and the action space for the task in the constructor, `__init__()`, and initialize any other variables you may need to run the task.

Now compare this with the first concrete task `Takeoff`, defined in `tasks/takeoff.py`:

```
class Takeoff(BaseTask):
    """Simple task where the goal is to lift off the ground and reach a ta
rget height."""
    ...
```

In `__init__()`, notice how the state and action spaces are defined using OpenAI Gym spaces (https://gym.openai.com/docs/#spaces), like `Box` (https://github.com/openai/gym/blob/master/gym/spaces/box.py). These objects provide a clean and powerful interface for agents to explore. For instance, they can inspect the dimensionality of a space (`shape`), ask for the limits (`high` and `low`), or even sample a bunch of observations using the `sample()` method, before beginning to interact with the environment. We also set a time limit (`max_duration`) for each episode here, and the height (`target_z`) that the quadcopter needs to reach for a successful takeoff.

The `reset()` method is meant to give you a chance to reset/initialize any variables you need in order to prepare for the next episode. You do not need to call it yourself; it will be invoked externally. And yes, it will be called once before each episode, including the very first one. Here `Takeoff` doesn't have any episode variables to initialize, but it must return a valid *initial condition* for the task, which is a tuple consisting of a `Pose` (http://docs.ros.org/api/geometry_msgs/html/msg/Pose.html) and `Twist` (http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html) object. These are ROS message types used to convey the pose (position, orientation) and velocity (linear, angular) you want the quadcopter to have at the beginning of an episode. You may choose to supply the same initial values every time, or change it a little bit, e.g. `Takeoff` drops the quadcopter off from a small height with a bit of randomness.

> **Tip**: Slightly randomized initial conditions can help the agent explore the state space faster.

Finally, the `update()` method is perhaps the most important. This is where you define the dynamics of the task and engage the agent. It is called by a ROS process periodically (roughly 30 times a second, by default), with current data from the simulation. A number of arguments are available: `timestamp` (you can use this to check for timeout, or compute velocities), `pose` (position, orientation of the quadcopter), `angular_velocity`, and `linear_acceleration`. You do not have to include all these variables in every task, e.g. `Takeoff` only uses pose information, and even that requires a 7-element state vector.

Once you have prepared the state you want to pass on to your agent, you will need to compute the reward, and check whether the episode is complete (e.g. agent crossed the time limit, or reached a certain height). Note that these two things (`reward` and `done`) are based on actions that the agent took in the past. When you are writing your own agents, you have to be mindful of this.

Now you can pass in the `state`, `reward` and `done` values to the agent's `step()` method and expect an action vector back that matches the action space that you have defined, in this case a `Box(6,)`. After checking that the action vector is non-empty, and clamping it to the space limits, you have to convert it into a ROS `Wrench` message. The first 3 elements of the action vector are interpreted as force in x, y, z directions, and the remaining 3 elements convey the torque to be applied around those axes, respectively.

Return the `Wrench` object (or `None` if you don't want to take any action) and the `done` flag from your `update()` method (note that when `done` is `True`, the `Wrench` object is ignored, so you can return `None` instead). This will be passed back to the simulation as a control command, and will affect the quadcopter's pose, orientation, velocity, etc. You will be able to gauge the effect when the `update()` method is called in the next time step.

Agents

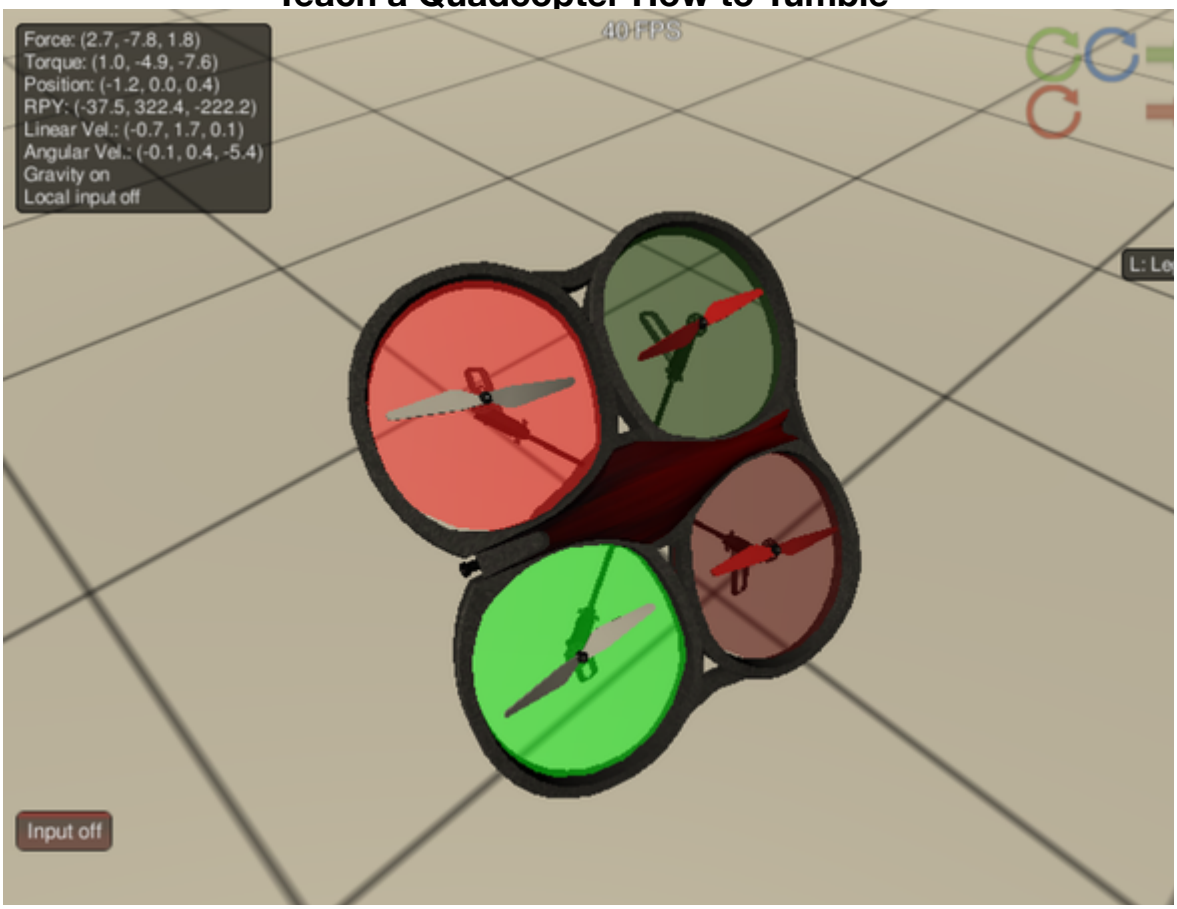
Reinforcement learning agents are defined in a similar way. Open up the generic agent class, `BaseAgent`, defined in `agents/base_agent.py`, and the sample agent `RandomPolicySearch` defined in `agents/policy_search.py`. They are actually even simpler to define - you only need to implement the `step()` method that is discussed above. It needs to consume `state` (vector), `reward` (scalar value) and `done` (boolean), and produce an `action` (vector). The state and action vectors must match the respective space indicated by the task. And that's it!

Well, that's just to get things working correctly! The sample agent given `RandomPolicySearch` uses a very simplistic linear policy to directly compute the action vector as a dot product of the state vector and a matrix of weights. Then, it randomly perturbs the parameters by adding some Gaussian noise, to produce a different policy. Based on the average reward obtained in each episode ("score"), it keeps track of the best set of parameters found so far, how the score is changing, and accordingly tweaks a scaling factor to widen or tighten the noise.

In [1]:

```
%%html
<div style="width: 100%; text-align: center;">
    <h3>Teach a Quadcopter How to Tumble</h3>
    <video poster="images/quadcopter_tumble.png" width="640" controls muted>
        <source src="images/quadcopter_tumble.mp4" type="video/mp4" />
        <p>Video: Quadcopter tumbling, trying to get off the ground</p>
    </video>
</div>
```

Teach a Quadcopter How to Tumble



Obviously, this agent performs very poorly on the task. It does manage to move the quadcopter, which is good, but instead of a stable takeoff, it often leads to dizzying cartwheels and somersaults! And that's where you come in - your first *task* is to design a better agent for this takeoff task. Instead of messing with the sample agent, create new file in the `agents/` directory, say `policy_gradients.py`, and define your own agent in it. Remember to inherit from the base agent class, e.g.:

```
class DDPG(BaseAgent):
    ...
```

You can borrow whatever you need from the sample agent, including ideas on how you might modularize your code (using helper methods like `act()`, `learn()`, `reset_episode_vars()`, etc.).

> **Note**: This setup may look similar to the common OpenAI Gym paradigm, but there is one small yet important difference. Instead of the agent calling a method on the environment (to execute an action and obtain the resulting state, reward and done value), here it is the task that is calling a method on the agent (`step()`). If you plan to store experience tuples for learning, you will need to cache the last state (S_{t-1}) and last action taken (A_{t-1}), then in the next time step when you get the new state (S_t) and reward (R_t), you can store them along with the `done` flag ($\langle S_{t-1}, A_{t-1}, R_t, S_t, \text{done?} \rangle$).

When an episode ends, the agent receives one last call to the `step()` method with `done` set to `True` - this is your chance to perform any cleanup/reset/batch-learning (note that no reset method is called on an agent externally). The action returned on this last call is ignored, so you may safely return `None`. The next call would be the beginning of a new episode.

One last thing - in order to run your agent, you will have to edit `agents/__init__.py` and import your agent class in it, e.g.:

```
from quad_controller_rl.agents.policy_gradients import DDPG
```

Then, while launching ROS, you will need to specify this class name on the commandline/terminal:

```
roslaunch quad_controller_rl rl_controller.launch agent:=DDPG
```

Okay, now the first task is cut out for you - follow the instructions below to implement an agent that learns to take off from the ground. For the remaining tasks, you get to define the tasks as well as the agents! Use the `Takeoff` task as a guide, and refer to the `BaseTask` docstrings for the different methods you need to override. Use some debug print statements to understand the flow of control better. And just like creating new agents, new tasks must inherit `BaseTask`, they need be imported into `tasks/__init__.py`, and specified on the commandline when running:

```
roslaunch quad_controller_rl rl_controller.launch task:=Hover agent:=DDPG
```

> **Tip**: You typically need to launch ROS and then run the simulator manually. But you can automate that process by either copying/symlinking your simulator to `quad_controller_rl/sim/DroneSim` (`DroneSim` must be an executable/link to one), or by specifying it on the command line, as follows:
>
> ```
> roslaunch quad_controller_rl rl_controller.launch task:=Hover
>  agent:=DDPG sim:=<full path>
> ```

Task 1: Takeoff

Implement takeoff agent

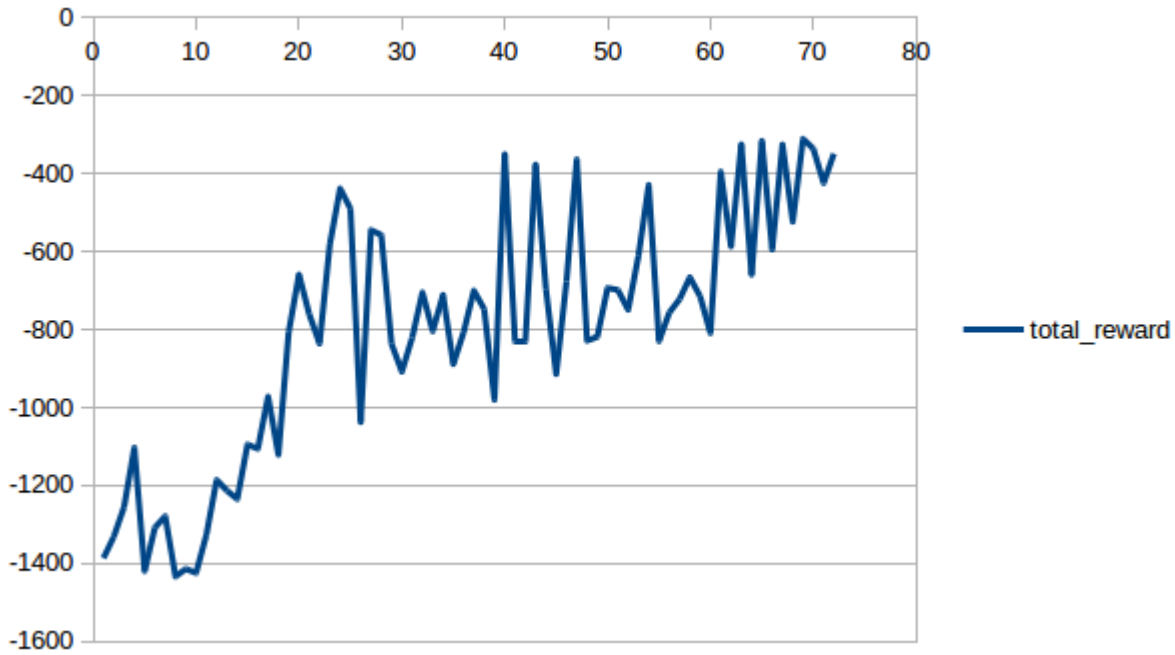
Train an agent to successfully lift off from the ground and reach a certain threshold height. Develop your agent in a file under `agents/` as described above, implementing at least the `step()` method, and any other supporting methods that might be necessary. You may use any reinforcement learning algorithm of your choice (note that the action space consists of continuous variables, so that may somewhat limit your choices).

The task has already been defined (in `tasks/takeoff.py`), which you should not edit. The default target height (Z-axis value) to reach is 10 units above the ground. And the reward function is essentially the negative absolute distance from that set point (upto some threshold). An episode ends when the quadcopter reaches the target height (x and y values, orientation, velocity, etc. are ignored), or when the maximum duration is crossed (5 seconds). See `Takeoff.update()` for more details, including episode bonus/penalty.

As you develop your agent, it's important to keep an eye on how it's performing. Build in a mechanism to log/save the total rewards obtained in each episode to file. Once you are satisfied with your agent's performance, return to this notebook to plot episode rewards, and answer the questions below.

Plot episode rewards

Plot the total rewards obtained in each episode, either from a single run, or averaged over multiple runs.



Q: What algorithm did you use? Briefly discuss why you chose it for this task.

A: I used Continuous control with deep reinforcement learning(DDPG) algorithm to train the quadcopter. It is a generalized algorithm whick can be used in any domain.It can be applied into the continuous domain and uses neural network for training which improves the perfomance of the system.

Q: Using the episode rewards plot, discuss how the agent learned over time.

A:

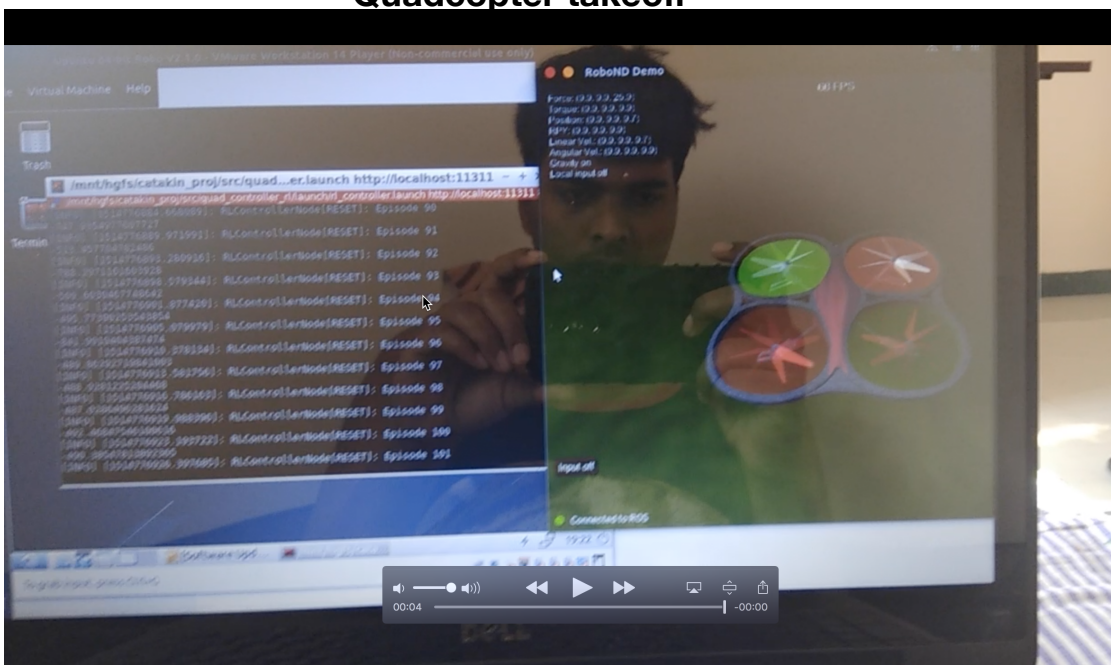
- It was a gradual learning, after 20 episodes it started flying(takeoff) to the max_height.
- The mean reward was around -400 at last.
- The learning was not too hard but the reward function was not that good for the proper takoff, without jerks and toppling.
- The reward function was structured such that the there was incentive for quadcopter to move up, but there was not any penality for the quadcopter not to toggle/spin. The reward function should contain orientation term to avoid toppling, rotatation of the quadcopter. As per the reward the quadcopter has learned over time how to go up, after few episodes the quadcopter achieves the maximum z-Force to go up, which is the optimal policy.
- Just by looking at the problem domain for take off one can deduce that only z-force plays the part. Rest all the parameters are not significant. So one can always do - x-force = 0 y-force = 0 z-force = predict only this parameter by neural network. x-torque = 0 y-torque = 0 z-torque = 0 Train the system for only predicting z-force.

  This the video when I trained the quadcopter for z-force only.

In [3]:

```
%%html
<div style="width: 100%; text-align: center;">
    <h3>Quadcopter takeoff</h3>
    <video poster="images/image.png" width="640" controls muted>
        <source src="images/takeoff.mp4" type="video/mp4" />
        <p>Video: Quadcopter tumbling, trying to get off the ground</p>
    </video>
</div>
```

Quadcopter takeoff



0:00 / 0:04

Task 2: Hover

Implement hover agent

Now, your agent must take off and hover at the specified set point (say, 10 units above the ground). Same as before, you will need to create an agent and implement the `step()` method (and any other supporting methods) to apply your reinforcement learning algorithm. You may use the same agent as before, if you think your implementation is robust, and try to train it on the new task. But then remember to store your previous model weights/parameters, in case your results were worth keeping.

States and rewards

Even if you can use the same agent, you will need to create a new task, which will allow you to change the state representation you pass in, how you verify when the episode has ended (the quadcopter needs to hover for at least a few seconds), etc. In this hover task, you may want to pass in the target height as part of the state (otherwise how would the agent know where you want it to go?). You may also need to revisit how rewards are computed. You can do all this in a new task file, e.g. `tasks/hover.py` (remember to follow the steps outlined above to create a new task):

```
class Hover(BaseTask):
    ...
```

Q: Did you change the state representation or reward function? If so, please explain below what worked best for you, and why you chose that scheme. Include short code snippet(s) if needed.

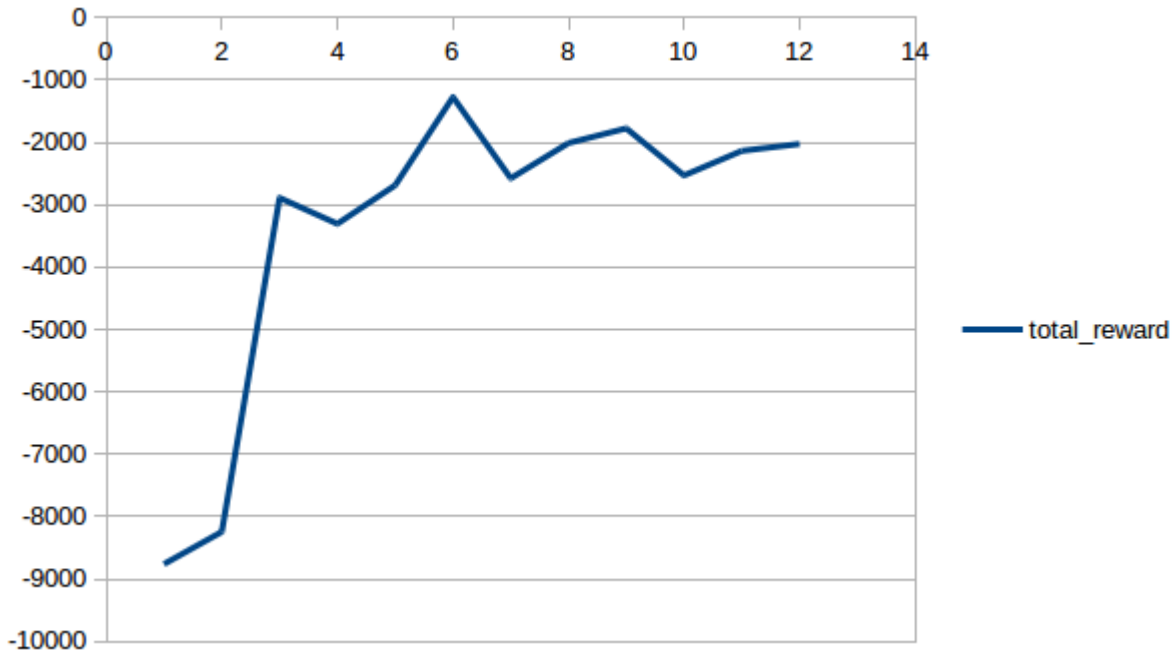
A: Yes I just tweaked the above reward function a little bit. I removed the condition where if max_height is attained, bonus reward is given and episode terminated. Now the quadcopter starts on a height of 'h' above the ground and some initial velocity to balance out the gravity.I also added penality term if the quadcopter crosses a fixed defined z-axis range.

Implementation notes

Q: Discuss your implementation below briefly, using the following questions as a guide:

- I used Continuous control with deep reinforcement learning(DDPG) algorithm to train the quadcopter.
- These are the hyper-parameters used for training.
  - actor learning rate = 0.0001
  - critic learning rate = 0.0001
  - gamma = 0.99
  - tau = 0.001
  - batch size = 64
- I used two neural networks.
  - Actor Network - 4 layered neural network. 2 hidden layer, one input and one output layer Layer1 - Number of nodes as size of state dimension. Layer2 - Fully connected layer with 400 nodes. Layer3 - Fully connected layer with 300 nodes. Layer4 - Output layer with size of action dimension.
  - Critic Network - 5 layered neural network. 3 hidden layer, one input and one output layer Layer1 - Take input from state and action in the first hidden layer. Layer2 - Fully connected layer with 400 nodes. Layer3 - Fully connected layer with 300 nodes. Layer4 - Fully connected layer with 300 nodes Layer5 - Output layer of 1 dimension.

Plot episode rewards



Task 3: Landing

What goes up, must come down! But safely!

Implement landing agent

This time, you will need to edit the starting state of the quadcopter to place it at a position above the ground (at least 10 units). And change the reward function to make the agent learn to settle down *gently*. Again, create a new task for this (e.g. `Landing` in `tasks/landing.py`), and implement the changes. Note that you will have to modify the `reset()` method to return a position in the air, perhaps with some upward velocity to mimic a recent takeoff.

Once you're satisfied with your task definition, create another agent or repurpose an existing one to learn this task. This might be a good chance to try out a different approach or algorithm.

Initial condition, states and rewards

Q: How did you change the initial condition (starting state), state representation and/or reward function? Please explain below what worked best for you, and why you chose that scheme. Were you able to build in a reward mechanism for landing gently?

A: I changed the starting state from (0,0,~0) to (0,0,h). 'h' is the height from which you want the quadcopter to start landing. I also gave some initial vecocity of 10 at the z axis so as to minimize the effect of gravity. The landing was not that gentle I expected, but it was good enough after few episodes. Reward function I used -

```
reward = -(abs(pose.position.z)*3 + abs(angular_velocity.x) + abs(angular_
velocity.y) + abs(angular_velocity.z) + abs(linear_acceleration.x) + abs(l
inear_acceleration.y) + abs(linear_acceleration.z) + abs(pose.orientation.
x) + abs(pose.orientation.y) + abs(pose.orientation.z)) # reward = zero fo
r matching target z, -ve as you go farther

 #Normalized the reward
 reward = reward / 4
 if pose.position.z <= 0:  # agent has crossed the target height
    reward += 10.0  # bonus reward
    done = True
 elif timestamp > self.max_duration:
 reward -= 10.0  # extra penalty
    done = True
```

I took terms of angular velocity,linear acceleration and pose into account.

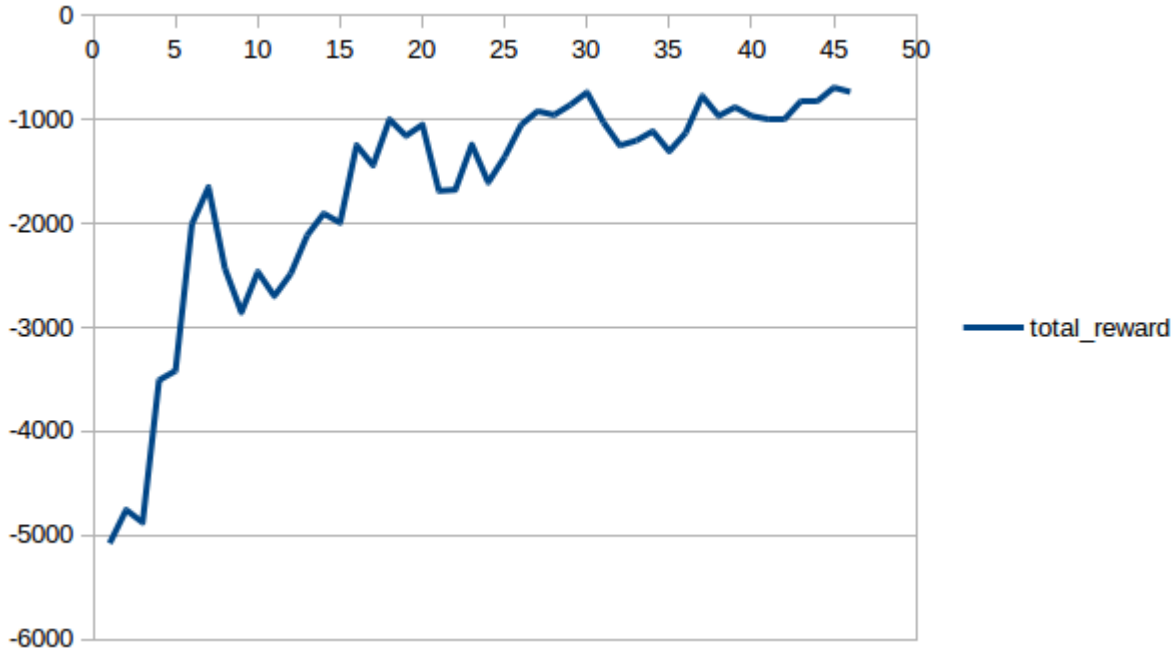
Implementation notes

Q: Discuss your implementation below briefly, using the same questions as before to guide you.

A: I took terms of angular velocity,linear acceleration and pose into account to build a reward function. And the policy I used was general enough to scale to this task also.

Plot episode rewards

As I have general policy which can run in multiple scenario's. The only task was to come up with a good reward function.



Task 4: Combined

In order to design a complete flying system, you will need to incorporate all these basic behaviors into a single agent.

Setup end-to-end task

The end-to-end task we are considering here is simply to takeoff, hover in-place for some duration, and then land. Time to create another task! But think about how you might go about it. Should it be one meta-task that activates appropriate sub-tasks, one at a time? Or would a single combined task with something like waypoints be easier to implement? There is no right or wrong way here - experiment and find out what works best (and then come back to answer the following).

Q: What setup did you ultimately go with for this combined task? Explain briefly.

A: Since the end-to-end task is open-ended. I used the same policy and changed only reward function to perform the final task.I used approaches to combine the rewards(takeoff,hover,landing) into one general reward.

Implement combined agent

Using your end-to-end task, implement the combined agent so that it learns to takeoff (at least 10 units above ground), hover (again, at least 10 units above ground), and gently come back to ground level.

Combination scheme and implementation notes

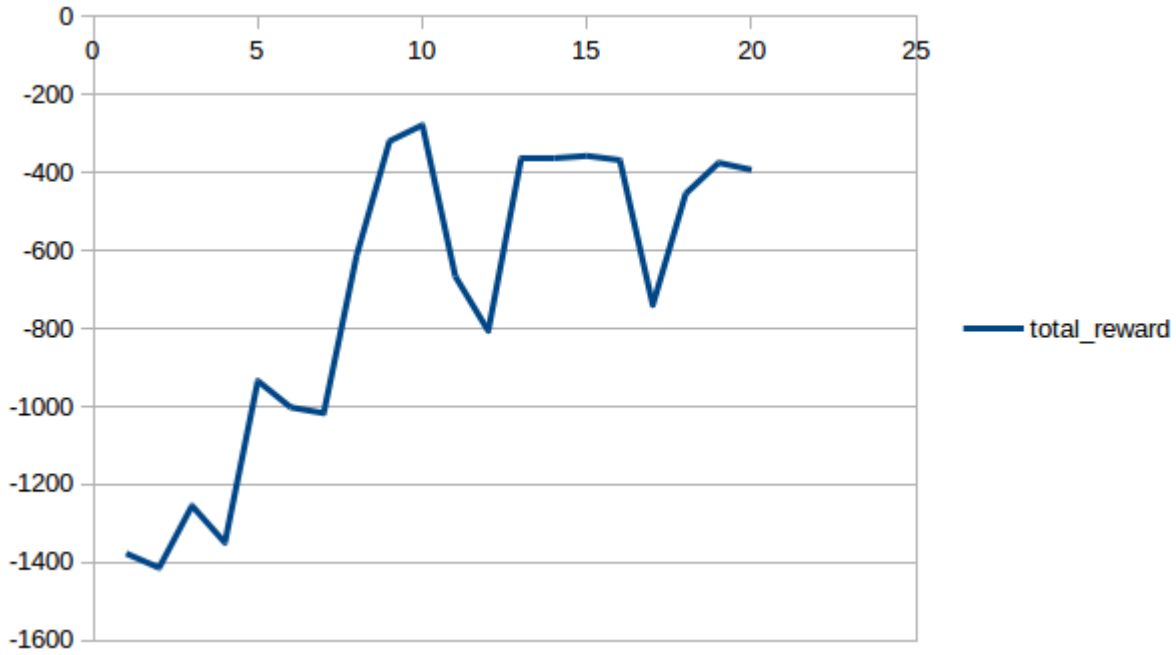
Just like the task itself, it's up to you whether you want to train three separate (sub-)agents, or a single agent for the complete end-to-end task.

Q: What did you end up doing? What challenges did you face, and how did you resolve them? Discuss any other implementation notes below.

A: For combinig the above 3 rewards into one generalized reward.I used 2 approaches.

- Normalizing all the 3 rewards and adding them into one single reward.
- Made reward function dynamic and dependent on time. I used the earlier 3 rewards fuctions based on time. For first 2sec - Takeoff reward, next 2sec- hover and at last landoff reward. Seocnd approach performed better than the first one. But still they didn't performed as expected. I also used the approach of training/predicting only z-axis force and rest making them 0(this knowledge I get from the problem domain), this approach gave very good results. I feel that there should be clarification given to the student wheter to use the problem domain knowledge or not in this project. Whether to make the agent generalized or specific to these tasks only.

Plot episode rewards


0
0
5
10
15
20
25
-200
-400
-600
-800
-1000
-1200
-1400
-1600
total_reward

Reflections

Q: Briefly summarize your experience working on this project. You can use the following prompts for ideas.

- What was the hardest part of the project? (e.g. getting started, running ROS, plotting, specific task, etc.)
- How did you approach each task and choose an appropriate algorithm/implementation for it?
- Did you find anything interesting in how the quadcopter or your agent behaved?

A: There was different chalenges at different steps. I enjoyed working on this project a lot. Also I learned a lot and understood the working of the ROS and its interactions with the simulator. I choose a generic algorithm which catered all my beeds, I used Continuous control with deep reinforcement learning(DDPG) algorithm to train the quadcopter. Initially the quadcopter did not know what to do, how to fly. But after few episodes it learnt what to do and how to fly.There was a lot of toppling and spinning in the first case as there was not any penalty in reward for not doing so. If we can include a penality term in the first reward function then that will solve the problem of toppling/spinning.

There are few **feedback** I want to provide -

- Setting up the ROS - People are facing some issues in setting up the ROS environment. I also faced some but was able to resolve it. Here are my some thoughts which I get form setting of the ROS environment. If we can put these into the README then that can help the students in dubuggin.
  - Make sure that you can ping the vm IP from the your Mac and Mac IP from vm.If the IP of vm is a.b.c.d . Run the command ping a.b.c.d and see if packets are being sent from both ways.
  - In the rossetting file. Make sure that both the flags are true.
- As pointed out earlier, just by looking at the problem domain for take off, hover, landing we can deduce that only z-force plays the part. Rest all the parameters are not significant. So one can always do -
  - x-force = 0
  - y-force = 0
  - z-force = predict only this parameter by neural network.
  - x-torque = 0
  - y-torque = 0
  - z-torque = 0
- Using this kind of neural network modelling one can ensure that the quadcopter does not topples and always move in z-direction.
- In README there should be more clarification regarding this, is the student allowed to use the domain knowledge and train the quadcopter or he should train quadcopter in a general fashion. Should he train for all the 6 parameters or he can train 1 parameter and keep all other 0 ?