RL-Quadcopter

October 15, 2018

1 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.

1.1 Instructions

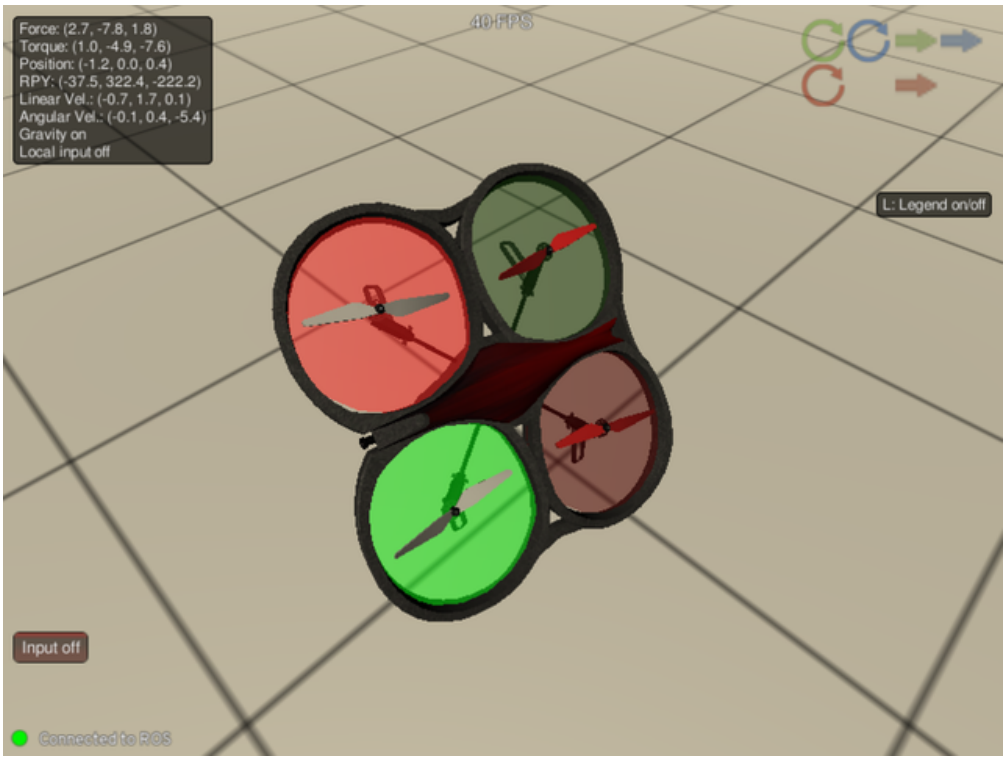
> **Note**: If you haven't done so already, follow the steps in this repo's README to install ROS, and ensure that the simulator is running and correctly connecting to ROS.

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.
- ...

1.1.1 Tasks

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



Quadcopter doing a flip trying to takeoff from the ground

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

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

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

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

    def update(self, timestamp, pose, angular_velocity, linear_acceleration):
        """Process current data, call agent, return action and done flag."""
        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 target height."""
    ...
```

In `__init__()`, notice how the state and action spaces are defined using OpenAI Gym spaces, like `Box`. 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` and `Twist` 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.

1.1.2 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>
```

```
<IPython.core.display.HTML object>
```

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 command-line/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>
> ```

1.2 Task 1: Takeoff

1.2.1 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.

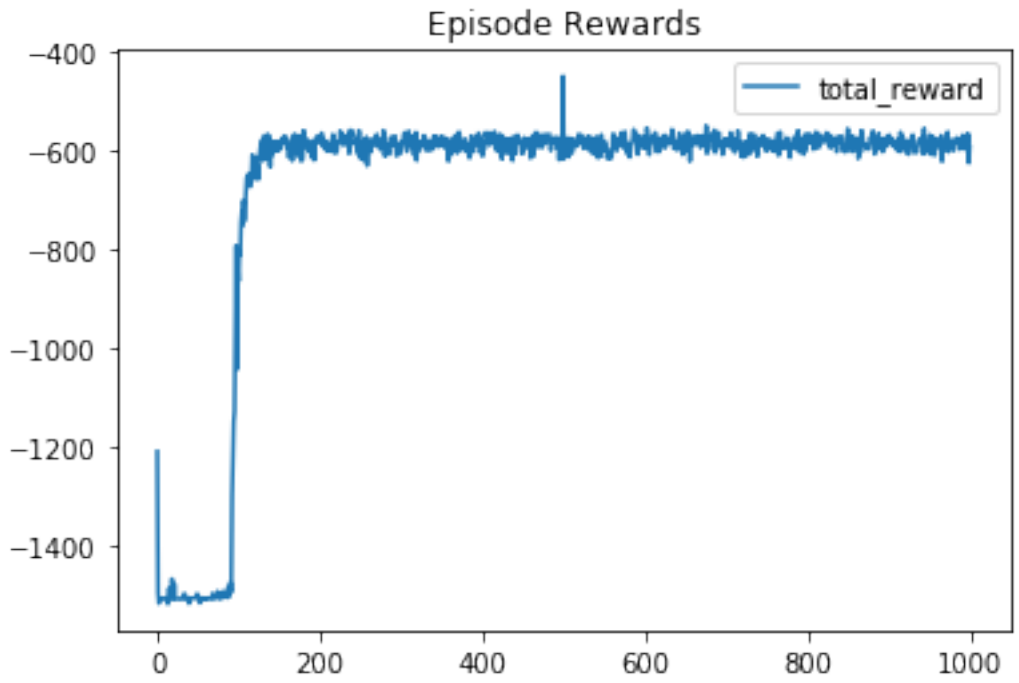
1.2.2 Plot episode rewards

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

```
In [302]: # TODO: Read and plot episode rewards
          %matplotlib inline
          import pandas as pd

          #df_stats = pd.read_csv('../out/task01/stats_2018-02-14_08-06-12.csv')
          df_stats = pd.read_csv('../out/task01/stats_2018-02-20_11-28-13.csv')
          df_stats[['total_reward']].plot(title="Episode Rewards")
```

```
Out[302]: <matplotlib.axes._subplots.AxesSubplot at 0x7fee0f7bd780>
```




```
In [15]: %%html
         <div style="width: 100%; text-align: center;">
             <h3>Teach a Quadcopter How to Takeoff</h3>
             <video poster="images/poster_01_takeoff.png" width="816" controls muted>
                 <source src="images/01_takeoff.mp4" type="video/mp4" />
                 <p>Video: Solution - Quadcopter Takeoff</p>
             </video>
         </div>
```

```
<IPython.core.display.HTML object>
```

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

A: The algorithm used was Deep Deterministic Policy Gradients (DDPG). This algorithm was chosen because of the continuous state and action spaces. This is actually an actor-critic method but the idea is similar. Alternatively, DQN (Deep Q-Network) could be used but the state and action spaces would have to converted to discrete space, an extra overhead I wanted to avoid.

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

- Was it an easy task to learn or hard?
- Was there a gradual learning curve, or an aha moment?
- How good was the final performance of the agent? (e.g. mean rewards over the last 10 episodes)

A: - Was it an easy task to learn or hard?

I think the answer to the question is dependant on the context. Once the setup was done, the model architecture was set, and the hyper-parameters were set, the task was not learned at all until around the episodes in the mid 100. All of a sudden, it learned the task.

However, what took up the most time was not the task itself but the overhead of getting a virtual machine of linux to run on my windows machine and get the network to communicate. I had to change BIOS settings to get this to work. It took 3 days to just get up and running. It would also crash all the time and use up all my memory. I think it took a week to even start the first task of take off. Then finding a good architecture (before it crashed) was challenging. In the end, having 3 layers of around 32 or 64 nodes worked. Did not need to user regularizers, dropout, or batch normalization.

- Was there a gradual learning curve, or an aha moment?

It was an aha momemnt. It didn't learn anything then it all of sudden learned around episode in the mid 100.

- How good was the final performance of the agent? (e.g. mean rewards over the last 10 episodes)

I would think they are good given the reward system used. I think they were good because the consistently stayed within the same range which was approximately -600. Please note the a reward system is arbitrarily set so good or bad depends on what the designer setup. This rewards system used negative rewards so the less negative the better. As you can see from my plot, it learned consistently around episode 200 but I let it run for 1000 episodes and the results were still consistent.

1.3 Task 2: Hover

1.3.1 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.

1.3.2 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 did change the state representation and reward function. I included velocity in the z-axis as a state input. Howere, the action space stayed the same. The reward system changed, it was a negative weighted sum of the error from the target height, target orientation, and the target velocity. I put the weight for orientation to 0.0 because orientation was not used. I biased more the weight for the position at 0.7 and 0.3 for the weight of the velocity. The more away from the target the actual value is the bigger the number. The sum is negative for a negative rewards (punishment.) So the more close to zero the less punishment. There is a punishment if the error for position is too high. There is a positive reward if the time lapsed surpassed the set duration.

1.3.3 Implementation notes

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

- What algorithm(s) did you try? What worked best for you?
- What was your final choice of hyperparameters (such as α, γ, ϵ, etc.)?
- What neural network architecture did you use (if any)? Specify layers, sizes, activation functions, etc.

A:

- What algorithm(s) did you try? What worked best for you?

The algorithm used was again Deep Deterministic Policy Gradients (DDPG). In the end about 2 or 3 layers with about 4 to 8 nodes per layer worked out. I think it worked out because I limited the state and action space to just one dimension in position and ignoring rotation. I actually solved the first task again with all 3 position axis but it took so long (and crashed all the time) I would never be able to finish the assignment. I added dropout and batch normalization because in general I find those work. I added saving out model weights early because it would keep crashing. I did not have to change the hyperparameters, just the model architecture and state and reward system.

- What was your final choice of hyperparameters (such as , , , etc.)?

I left the hyperameters as they initially were in the Takeoff task. I changed the model architecture and the state and reward system to get the results I was looking for. I would like to point out that I was stuck on this task for about 2 weeks but then Udacity added a 12.Troubleshooting

section. There I saw that you were allowed to change the initial height to the target height. Before, I was trying to train it to takeoff to target height and then hover. It would keep crashing around 200 episodes or 1000 episodes so I couldn't really train it even if I wanted to.

- What neural network architecture did you use (if any)? Specify layers, sizes, activation functions, etc.

Th neural network architecture used was about 2 or 3 layers with about 4 to 8 nodes per layer for both the actor and the critic. I also added dropout and batch normalization. I found using less nodes and in general dropout and batch normalization helped out. Activation functions remained rectified linear units (relu) for all hidden layers. I did try sigmoid for all the hidden layers, I did not notice any difference visually or in the plot so I went back to relu.

1.3.4 Plot episode rewards

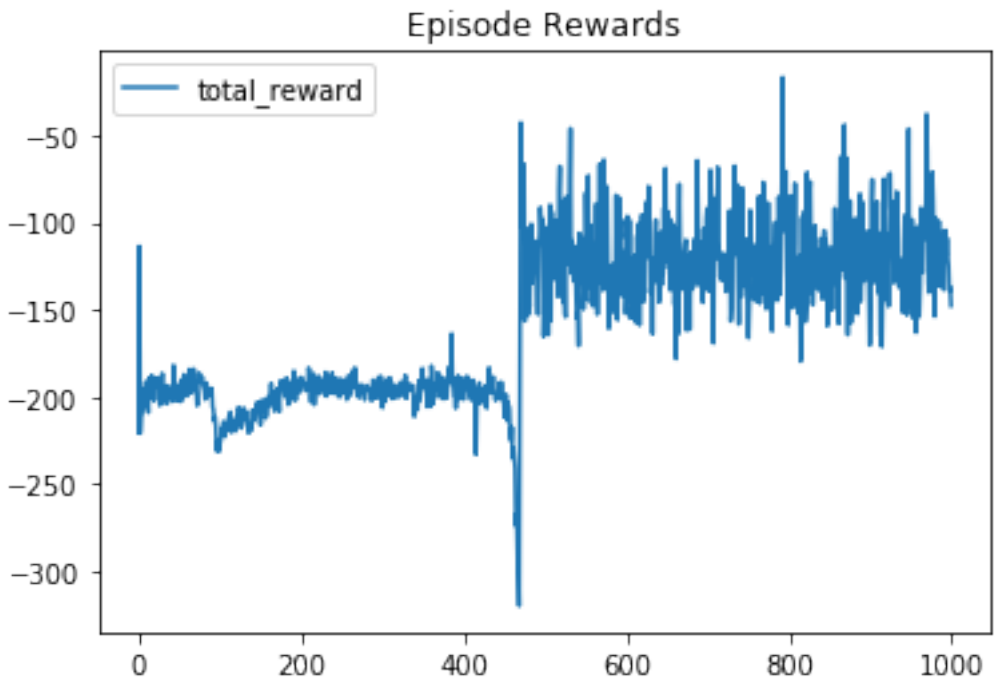
As before, plot the episode rewards, either from a single run, or averaged over multiple runs. Comment on any changes in learning behavior.

```
In [313]: # TODO: Read and plot episode rewards
          %matplotlib inline
          import pandas as pd

          # stats_2018-02-19_15-18-18.csv - Started to work around episode 650
          # stats_2018-02-20_10-32-16 - Should have been good around episode 350 but overrode
          # This should work stats_2018-02-20_11-02-40.csv.
          df_stats = pd.read_csv('../out/task02/stats_2018-02-20_12-43-46.csv')

          #df_stats = pd.read_csv('../out/task02/stats_2018-02-20_08-13-41.csv')
          df_stats[['total_reward']].plot(title="Episode Rewards")

Out[313]: <matplotlib.axes._subplots.AxesSubplot at 0x7fee0f500470>
```




```
In [17]: %%html
         <div style="width: 100%; text-align: center;">
             <h3>Teach a Quadcopter How to Hover</h3>
             <video poster="images/poster_02_hover.png" width="816" controls muted>
                 <source src="images/02_hover.mp4" type="video/mp4" />
                 <p>Video: Solution - Quadcopter Hover</p>
             </video>
         </div>
```

```
<IPython.core.display.HTML object>
```

1.4 Task 3: Landing

What goes up, must come down! But safely!

1.4.1 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.

1.4.2 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:

The initial condition was changed to start at the target height. State representation is the same as the Hover task, position z and velocity z. The reward function changed. It is similar in idea as the hover reward system except that I changed it to be a positive reward system and exponential. I only use position z and velocity z. It gets rewards by having the value close to the target. This is done by subtracting from 1 the difference and then taking the power (in this case 2). For example if the position is at the target, the difference would be 0. Subtracting 0 from 1 would make it 1 and then squaring it. So you get more points for having a difference closer to zero. I also gave a bonus for landing gently. If the velocity was too high, I gave a negative reward (punishment,) and end the episode. This way, the quadcopter cannot finish the episode if it is moving too fast.

1.4.3 Implementation notes

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

A:

-What algorithm(s) did you try? What worked best for you? I still used DDPG again. I just changed the reward system. The model architecture was the same as hover.

-What was your final choice of hyperparameters (such as , , , etc.)? I used the same hyperparameters as the hover and the takeoff. I just changed the architecture.

-What neural network architecture did you use (if any)? Specify layers, sizes, activation functions, etc. I kept the same network architecture as hover. Th neural network architecture used was about 2 or 3 layers with about 4 to 8 nodes per layer for both the actor and the critic. I also added dropout and batch normalization. I found using less nodes and in general dropout and batch normalization helped out. Activation functions remained rectified linear units (relu) for all hidden layers.

1.4.4 Plot episode rewards

As before, plot the episode rewards, either from a single run, or averaged over multiple runs. This task is a little different from the previous ones, since you're starting in the air. Was it harder to learn? Why/why not?

```
In [426]: # TODO: Read and plot episode rewards
          %matplotlib inline
          import pandas as pd

          df_stats = pd.read_csv('../out/task03/stats_2018-02-20_21-03-45.csv')

          df_stats[['total_reward']].plot(title="Episode Rewards")
```

```
Out[426]: <matplotlib.axes._subplots.AxesSubplot at 0x7fee0d7bd0b8>
```


```
In [18]: %%html
         <div style="width: 100%; text-align: center;">
             <h3>Teach a Quadcopter How to Land</h3>
             <video poster="images/poster_03_land.png" width="816" controls muted>
                 <source src="images/03_land.mp4" type="video/mp4" />
                 <p>Video: Solution - Quadcopter Land</p>
             </video>
         </div>
```

```
<IPython.core.display.HTML object>
```

1.5 Task 4: Combined

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

1.5.1 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:

1.5.2 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.

1.5.3 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: What I ended up doing was created an agent for task 4 called task04_ddpg_agent.py with a class called Task04_DDPG. I also created a task called combined.py with a class called Combined. How these worked is by using instances of the other tasks object from classes and swapped them depending on the mode. The swapping used a modulus

self.mode %= 3

and the possible modes are 0, 1, 2. Where mode 0 is task takeoff, mode 1 is task hover, and mode 2 is land. The task would just keep cycling if the simulator is running.

For the agent, in the constructor **init**, the 3 task objects and 3 agents were instantiated and assigned to member variables. The instance for the task is used to instantiate the appropriate instance of the corresponding agent. This was important because numpy shapes and member variables would not match up if they were mixed and matched. The model weights for each tasks were trained and preloaded from the other tasks. You can look at the different model weights and plots generated for this project in the 'out' folder under its corresponding task.s

```
    self.task_takeoff = takeoff_b.TakeoffB()
    self.task_hover = hover_b.HoverB()
    self.task_land = land_b.LandB()
    self.o_task01_agent = task01_ddpg_agent_b.Task01_DDPG(self.task_takeoff)
    self.o_task02_agent = task02_ddpg_agent_b.Task02_DDPG(self.task_hover)
    self.o_task03_agent = task03_ddpg_agent_b.Task03_DDPG(self.task_land)
```

Then to know which agent instance to use, a member variable would be set. Depending on the mode, this member variable would be updated.

```
    # Current agent
    self.o_current_agent = self.o_task01_agent
```

For the agents all the same methods exists so they would be called, the instance method would be called and returned.

```
def act(self, states):
    return self.o_current_agent.act(states)
```

To know when to swap instances for agents, the 'done' variable was checked during the step() method.

```
def step(self, state, reward, done):

    self.total_reward += reward

    if done:




        # DEBUG
        #
        print("\n\nDEBUG - done have been called. self.mode: {}\n\n".format(self.mode))

        # Go to the mode
        self.mode += 1
        #
        # Cycle mode back to the beginning
        self.mode %= 3

        if self.mode == 0:
            self.o_current_agent = self.o_task01_agent
            self.task  = self.task_takeoff

        if self.mode == 1:
            self.o_current_agent = self.o_task02_agent
            self.task =self.task_hover

        if self.mode == 2:
            self.o_current_agent = self.o_task03_agent
            self.task = self.task_land

        done = False

        # If we cycle back to take off, we count it as an episode
        # and reset reward.
        if self.mode == 0:
            # Write episode stats
            self.write_stats([self.episode_num, self.total_reward])

            self.reset_episode_vars()
            self.episode_num += 1

    return self.o_current_agent.step(state, reward, done)
```

NOTE: self.task was not used in the end because I instantiated the agents with their correct task instances in the **init**, but I kept it there.

This alone almost worked. When 'done' is True, .reset() method for a task is called. However,

I do not want the position or the angular or linear velocity to be reset. I created a new member variable to all of the tasks called

```
self.current_condition
```

In the task Combine class, in the update() method takes the current condition and set it as the member variable for the current task

```
def update(self, timestamp, pose, angular_velocity, linear_acceleration):

    # Save current condition
    position = np.array([pose.position.x, pose.position.y, pose.position.z])
    self.current_condition = Pose(
                position=Point(*position),
                orientation=Quaternion(0.0, 0.0, 0.0, 1.0),
            ), Twist(
                linear=linear_acceleration,
                angular=angular_velocity
            )

    # Update the tasks current condition.
    self.o_current_task.current_condition = self.current_condition
```

When 'done' is set the True in the Combine class method .update(), the mode will swap the current task it will also set the newly set current condition to be what it is now and not reset it to a height of zero or target height and still have the same angular and linear velocity.

```
    if done:
        self.mode += 1
        self.mode %= 3

        if self.mode == 0:
            self.o_current_task = self.task_takeoff

        if self.mode == 1:
            self.o_current_task = self.task_hover

        if self.mode == 2:
            self.o_current_task = self.task_land

        done = False

    # Update the tasks current condition.
    self.o_current_task.current_condition = self.current_condition
```

If you let the simulation run, it will just loop.

The reward is given by a combine specific episode. This is anytime the mode loops back to 0. The rewards are added up and written to the .csv.

I was going to freeze all the layers for the models to prevent training, but decided not to.

NOTE: It seems that for all the simulations (Takeoff, Hover, Land, Combined) the first episode is useless. The second episode and beyond works. I don't know how to prevent this. I gather that this is expected behaviour.

1.5.4 Plot episode rewards

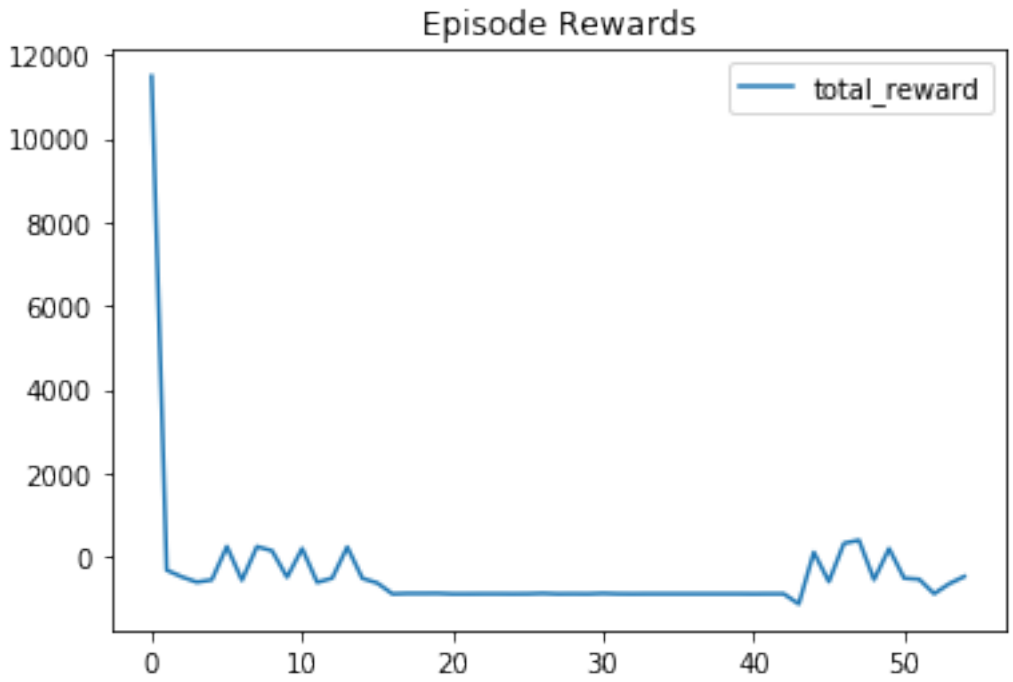
As before, plot the episode rewards, either from a single run, or averaged over multiple runs.

```
In [433]: # TODO: Read and plot episode rewards
          %matplotlib inline
          import pandas as pd

          df_stats = pd.read_csv('../out/task04/stats_2018-02-21_14-07-26.csv')

          df_stats[['total_reward']].plot(title="Episode Rewards")
```

```
Out[433]: <matplotlib.axes._subplots.AxesSubplot at 0x7fee0d61d470>
```




```
In [19]: %%html
         <div style="width: 100%; text-align: center;">
             <h3>Teach a Quadcopter How to combined Takeoff, Hover, and Land</h3>
             <video poster="images/poster_04_combined.png" width="816" controls muted>
                 <source src="images/04_combined.mp4" type="video/mp4" />
                 <p>Video: Solution - Quadcopter combine Takeoff, Hover, and Land</p>
```

```
            </video>
        </div>

<IPython.core.display.HTML object>
```

1.6 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:

- What was the hardest part of the project? (e.g. getting started, running ROS, plotting, specific task, etc.)

Getting setup was by far the hardest thing about this project. I could not get it run in the cloud. It used up all my computer's resources. I had to change BIOS settings. I had to use 2 different operating systems. The simulation crashed. Then the tasks were challenging themselves because there were no examples. Luckily, there were a few dedicated people trying to complete the assignment in Slack where I can bounce ideas off of.

Unfortunately, I had a lot of prior commitment so it was hard to give the dedicated extra time to get this assignment done. I had to stay up until 3 or 4 am in the morning and a lot of the times the simulations were worthless. I also wasn't sure if I was even on the right track.

Also, the office hours I have on my calendar were for general deep learning questions. There were very few office hours specifically for this assignment. this made it hard as well.

- How did you approach each task and choose an appropriate algorithm/implementation for it?

I stuck with DDPG mainly because the problem was continuous. The lectures taught with discrete space, but DDPG was the only one explained with continuous space. I just stuck with it. I know other people solved it using DQN. I did not want the extra overhead of converting continuous space to discrete space. I had enough hoops to jump through.

- Did you find anything interesting in how the quadcopter or your agent behaved?

The implementation I did seemed too abrupt. I would rather a smoother transition. I would also like to know what are waypoints and how to use them. I assume waypoints may solve the abrupt problem. I also used different model architecture and state spaces for the tasks. I figured if I was able to use the same model architecture and state space for all the tasks, I would just have to update the weights of the model and transitions between tasks would be smoother. The code would be easier to write as well instead of swapping current tasks and current agents.

1.6.1 To run all 4 simulation

NOTE: It seems that for all the simulations (Takeoff, Hover, Land, Combined) the first episode is useless. The second episode and beyond works. I don't know how to prevent this. I gather that this is expected behaviour.

```
roslaunch quad_controller_rl rl_controller.launch task:=Takeoff agent:=Task01_DDPG
roslaunch quad_controller_rl rl_controller.launch task:=Hover agent:=Task02_DDPG
roslaunch quad_controller_rl rl_controller.launch task:=Land agent:=Task03_DDPG
roslaunch quad_controller_rl rl_controller.launch task:=Combined agent:=Task04_DDPG
```