Project: Train a Quadcopter How to Fly

Design an agent to 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

Take a look at the files in the directory to better understand the structure of the project.

- `task.py` : Define your task (environment) in this file.
- `agents/` : Folder containing reinforcement learning agents.
  - `policy_search.py` : A sample agent has been provided here.
  - `agent.py` : Develop your agent here.
- `physics_sim.py` : This file contains the simulator for the quadcopter. **DO NOT MODIFY THIS FILE**.

For this project, you will define your own task in `task.py` . Although we have provided a example task to get you started, you are encouraged to change it. Later in this notebook, you will learn more about how to amend this file.

You will also design a reinforcement learning agent in `agent.py` to complete your chosen task.

You are welcome to create any additional files to help you to organize your code. For instance, you may find it useful to define a `model.py` file defining any needed neural network architectures.

Controlling the Quadcopter

We provide a sample agent in the code cell below to show you how to use the sim to control the quadcopter. This agent is even simpler than the sample agent that you'll examine (in `agents/policy_search.py`) later in this notebook!

The agent controls the quadcopter by setting the revolutions per second on each of its four rotors. The provided agent in the `Basic_Agent` class below always selects a random action for each of the four rotors. These four speeds are returned by the `act` method as a list of four floating-point numbers.

For this project, the agent that you will implement in `agents/agent.py` will have a far more intelligent method for selecting actions!

In [1]:
```
import random

class Basic_Agent():
    def __init__(self, task):
        self.task = task

    def act(self):
        new_thrust = random.gauss(450., 25.)
        return [new_thrust + random.gauss(0., 1.) for x in range(4)]
```

Run the code cell below to have the agent select actions to control the quadcopter.

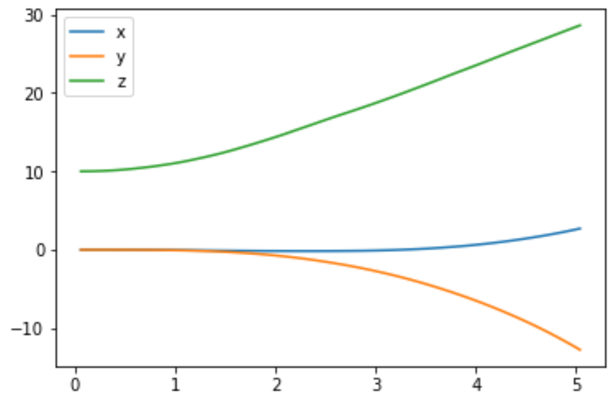
Feel free to change the provided values of `runtime`, `init_pose`, `init_velocities`, and `init_angle_velocities` below to change the starting conditions of the quadcopter.

The `labels` list below annotates statistics that are saved while running the simulation. All of this information is saved in a text file `data.txt` and stored in the dictionary `results`.

In [2]:
```
%load_ext autoreload
%autoreload 2

import csv
import numpy as np
from task import Task

# Modify the values below to give the quadcopter a different starting position.
runtime = 5.                                     # time limit of the episode
init_pose = np.array([0., 0., 10., 0., 0., 0.])  # initial pose
init_velocities = np.array([0., 0., 0.])         # initial velocities
init_angle_velocities = np.array([0., 0., 0.])   # initial angle velocities
file_output = 'data.txt'                         # file name for saved results

# Setup
task = Task(init_pose, init_velocities, init_angle_velocities, runtime)
agent = Basic_Agent(task)
done = False
labels = ['time', 'x', 'y', 'z', 'phi', 'theta', 'psi', 'x_velocity',
          'y_velocity', 'z_velocity', 'phi_velocity', 'theta_velocity',
          'psi_velocity', 'rotor_speed1', 'rotor_speed2', 'rotor_speed3', 'roto
r_speed4']
results = {x : [] for x in labels}

# Run the simulation, and save the results.
with open(file_output, 'w') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(labels)
    while True:
        rotor_speeds = agent.act()
        _, _, done = task.step(rotor_speeds)
        to_write = [task.sim.time] + list(task.sim.pose) + list(task.sim.v) + l
ist(task.sim.angular_v) + list(rotor_speeds)
        for ii in range(len(labels)):
            results[labels[ii]].append(to_write[ii])
        writer.writerow(to_write)
        if done:
            break
```

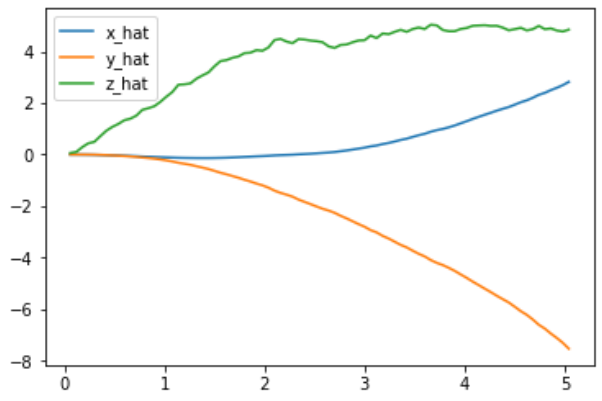
Run the code cell below to visualize how the position of the quadcopter evolved during the simulation.

In [3]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(results['time'], results['x'], label='x')
plt.plot(results['time'], results['y'], label='y')
plt.plot(results['time'], results['z'], label='z')
plt.legend()
_ = plt.ylim()
```



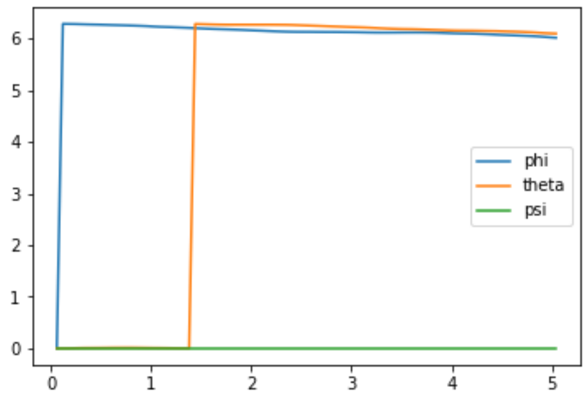
The next code cell visualizes the velocity of the quadcopter.

In [4]:
```
plt.plot(results['time'], results['x_velocity'], label='x_hat')
plt.plot(results['time'], results['y_velocity'], label='y_hat')
plt.plot(results['time'], results['z_velocity'], label='z_hat')
plt.legend()
_ = plt.ylim()
```



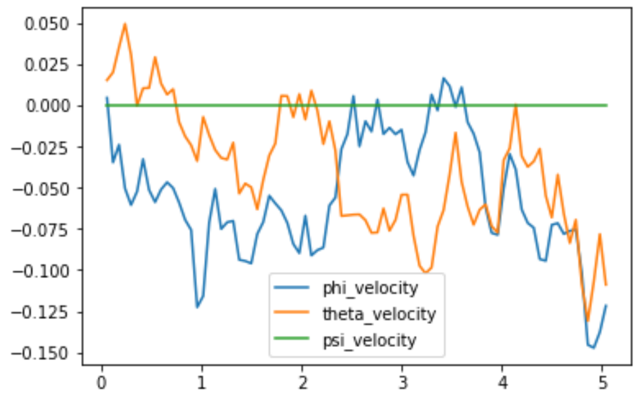
Next, you can plot the Euler angles (the rotation of the quadcopter over the x-, y-, and z-axes),

In [5]:
```
plt.plot(results['time'], results['phi'], label='phi')
plt.plot(results['time'], results['theta'], label='theta')
plt.plot(results['time'], results['psi'], label='psi')
plt.legend()
_ = plt.ylim()
```



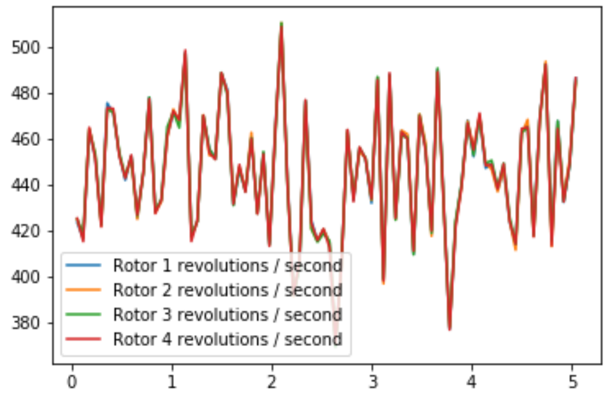
before plotting the velocities (in radians per second) corresponding to each of the Euler angles.

In [6]:
```
plt.plot(results['time'], results['phi_velocity'], label='phi_velocity')
plt.plot(results['time'], results['theta_velocity'], label='theta_velocity')
plt.plot(results['time'], results['psi_velocity'], label='psi_velocity')
plt.legend()
_ = plt.ylim()
```



Finally, you can use the code cell below to print the agent's choice of actions.

In [7]:
```
plt.plot(results['time'], results['rotor_speed1'], label='Rotor 1 revolutions / second')
plt.plot(results['time'], results['rotor_speed2'], label='Rotor 2 revolutions / second')
plt.plot(results['time'], results['rotor_speed3'], label='Rotor 3 revolutions / second')
plt.plot(results['time'], results['rotor_speed4'], label='Rotor 4 revolutions / second')
plt.legend()
_ = plt.ylim()
```



When specifying a task, you will derive the environment state from the simulator. Run the code cell below to print the values of the following variables at the end of the simulation:

- `task.sim.pose` (the position of the quadcopter in (x, y, z) dimensions and the Euler angles),
- `task.sim.v` (the velocity of the quadcopter in (x, y, z) dimensions), and
- `task.sim.angular_v` (radians/second for each of the three Euler angles).

In [8]:
```
# the pose, velocity, and angular velocity of the quadcopter at the end of the episode
print(task.sim.pose)
print(task.sim.v)
print(task.sim.angular_v)
```

```
[  2.70782005 -12.74256569  28.60573449   6.0142624    6.09433561   0.        ]
[ 2.82232525 -7.53553576  4.84810677]
[-0.12177709 -0.10893043  0.        ]
```

In the sample task in `task.py` , we use the 6-dimensional pose of the quadcopter to construct the state of the environment at each timestep. However, when amending the task for your purposes, you are welcome to expand the size of the state vector by including the velocity information. You can use any combination of the pose, velocity, and angular velocity - feel free to tinker here, and construct the state to suit your task.

The Task

A sample task has been provided for you in `task.py` . Open this file in a new window now.

The `__init__()` method is used to initialize several variables that are needed to specify the task.

- The simulator is initialized as an instance of the `PhysicsSim` class (from `physics_sim.py`).
- Inspired by the methodology in the original DDPG paper, we make use of action repeats. For each timestep of the agent, we step the simulation `action_repeats` timesteps. If you are not familiar with action repeats, please read the **Results** section in the DDPG paper (https://arxiv.org/abs/1509.02971).
- We set the number of elements in the state vector. For the sample task, we only work with the 6-dimensional pose information. To set the size of the state (`state_size`), we must take action repeats into account.
- The environment will always have a 4-dimensional action space, with one entry for each rotor (`action_size=4`). You can set the minimum (`action_low`) and maximum (`action_high`) values of each entry here.
- The sample task in this provided file is for the agent to reach a target position. We specify that target position as a variable.

The `reset()` method resets the simulator. The agent should call this method every time the episode ends. You can see an example of this in the code cell below.

The `step()` method is perhaps the most important. It accepts the agent's choice of action `rotor_speeds` , which is used to prepare the next state to pass on to the agent. Then, the reward is computed from `get_reward()` . The episode is considered done if the time limit has been exceeded, or the quadcopter has travelled outside of the bounds of the simulation.

In the next section, you will learn how to test the performance of an agent on this task.

The Agent

The sample agent given in `agents/policy_search.py` 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.

Run the code cell below to see how the agent performs on the sample task.

In [9]:
```
import sys
import pandas as pd
from agents.policy_search import PolicySearch_Agent
from task import Task

num_episodes = 1000
target_pos = np.array([0., 0., 10.])
task = Task(target_pos=target_pos)
agent = PolicySearch_Agent(task)

for i_episode in range(1, num_episodes+1):
    state = agent.reset_episode() # start a new episode
    while True:
        action = agent.act(state)
        next_state, reward, done = task.step(action)
        agent.step(reward, done)
        state = next_state
        if done:
            print("\rEpisode = {:4d}, score = {:7.3f} (best = {:7.3f}), noise_s
cale = {}".format(
                i_episode, agent.score, agent.best_score, agent.noise_scale), e
nd="")  # [debug]
            break
    sys.stdout.flush()
```

```
Episode = 1000, score =  -0.323 (best =  -0.055), noise_scale = 3.25
```

This agent should perform very poorly on this task. And that's where you come in!

Define the Task, Design the Agent, and Train Your Agent!

Amend `task.py` to specify a task of your choosing. If you're unsure what kind of task to specify, you may like to teach your quadcopter to takeoff, hover in place, land softly, or reach a target pose.

After specifying your task, use the sample agent in `agents/policy_search.py` as a template to define your own agent in `agents/agent.py` . 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()` , etc.).

Note that it is **highly unlikely** that the first agent and task that you specify will learn well. You will likely have to tweak various hyperparameters and the reward function for your task until you arrive at reasonably good behavior.

As you develop your agent, it's important to keep an eye on how it's performing. Use the code above as inspiration to build in a mechanism to log/save the total rewards obtained in each episode to file. If the episode rewards are gradually increasing, this is an indication that your agent is learning.

I consolidated the DDPG Agent and task into one function. This allows me to control all hyperparameters of quadcopter.

In [10]:
```
# Import the Consolidated Agent Function
from agents.agent import QuadCopter_Train
```

```
Using TensorFlow backend.
```

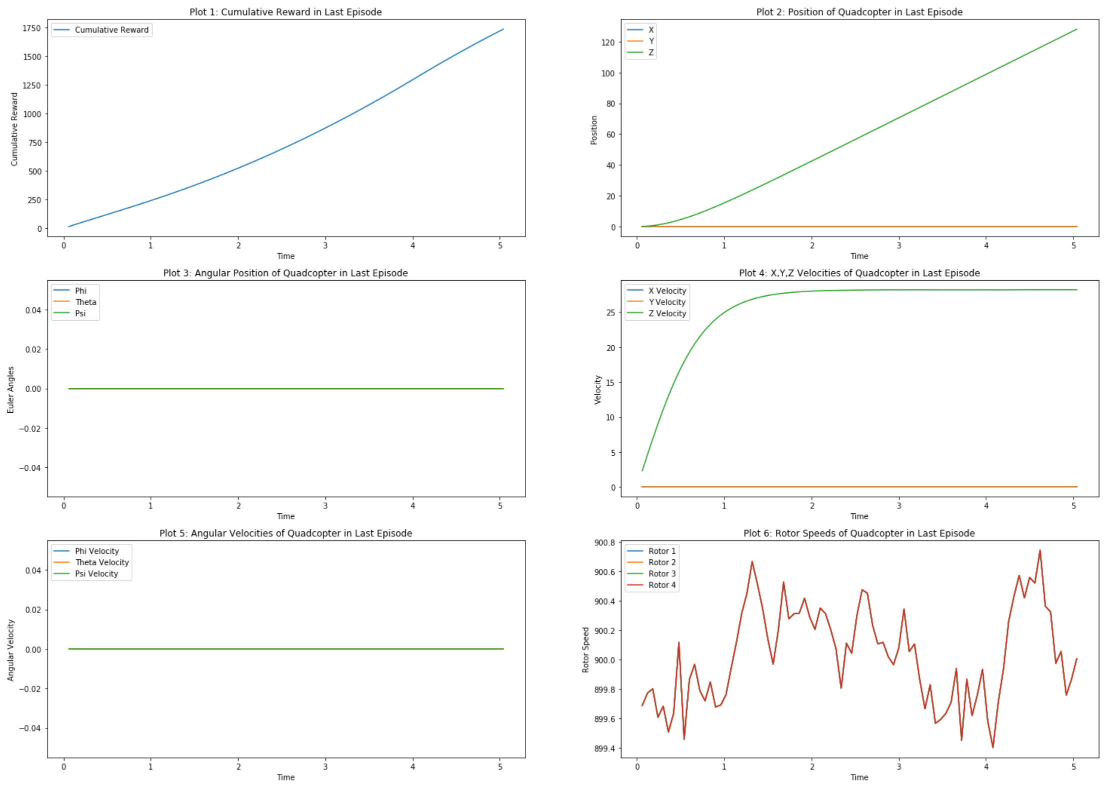
In [11]:
```
Results = QuadCopter_Train(Num_Episodes = 100,
                           Action_Size = 1,
                           Action_High = 900,
                           Target_Pos = np.array([0., 0., 100.]),
                           Reward = '10 - 0.05 * abs(self.sim.pose[2] - self.ta
rget_pos[2]) - 0.03*abs(self.sim.v[2])')
```

```
Episode =  100,     Cumulative_Reward =  1732.826     (Best =  1733.022)
Quadcopter Trained!
```

Plot the Rewards

Once you are satisfied with your performance, plot the episode rewards, either from a single run, or averaged over multiple runs.

In [12]:
```
## TODO: Plot the rewards.
Last_Episode_Run = Results.loc[Results.episode == 100]

# Plot the Results
import matplotlib.pyplot as plt

# Make a Subplt figure
plt.subplots(figsize=(25,18))

# Plot 1: Cumulative Reward in Simulation
plt.subplot(321)
plt.plot(Last_Episode_Run.time,Last_Episode_Run.cumulative_reward,label ='Cumul
ative Reward')
plt.xlabel('Time')
plt.ylabel('Cumulative Reward')
plt.title("Plot 1: Cumulative Reward in Last Episode")
plt.legend(loc = 2)

# Plot 2: Position (X,Y,Z)
plt.subplot(322)
plt.plot(Last_Episode_Run.time,Last_Episode_Run.x,label="X")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.y,label = "Y")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.z, label = "Z")
plt.xlabel('Time')
plt.ylabel('Position')
plt.title("Plot 2: Position of Quadcopter in Last Episode")
plt.legend(loc = 2)

# Plot 3: Position or Orientation (Phi, Thera, Psi: Euler Angles)
plt.subplot(323)
plt.plot(Last_Episode_Run.time,Last_Episode_Run.phi,label="Phi")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.theta,label = "Theta")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.psi, label = "Psi")
plt.xlabel('Time')
plt.ylabel('Euler Angles')
plt.title("Plot 3: Angular Position of Quadcopter in Last Episode")
plt.legend(loc = 2)

# Plot 4: Velocites
plt.subplot(324)
plt.plot(Last_Episode_Run.time,Last_Episode_Run.x_velocity,label="X Velocity")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.y_velocity,label = "Y Velocity"
)
plt.plot(Last_Episode_Run.time,Last_Episode_Run.z_velocity, label = "Z Velocity
")
plt.xlabel('Time')
plt.ylabel('Velocity')
plt.title("Plot 4: X,Y,Z Velocities of Quadcopter in Last Episode")
plt.legend(loc = 2)

# Plot 5: Angular Velocites
plt.subplot(325)
plt.plot(Last_Episode_Run.time,Last_Episode_Run.phi_velocity,label="Phi Velocit
y")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.theta_velocity,label = "Theta V
elocity")
plt.plot(Last_Episode_Run.time,Last_Episode_Run.psi_velocity, label = "Psi Velo
city")
plt.xlabel('Time')
plt.ylabel('Angular Velocity')
plt.title("Plot 5: Angular Velocities of Quadcopter in Last Episode")
plt.legend(loc = 2)
```



Reflections

Question 1: Describe the task that you specified in `task.py` . How did you design the reward function?

Answer: I created a simple task of take off and hovering at a height (z) of 100. The reward function is also pretty straight forward. For this task, only the a position matters so the reward function penalizes any deviation from z position and ignores x and y position completely. if i add x and y positions to the mix, then the feedback signal has too much noise and the agent fails to converge to a good solution.

Next, i also added a penalty for z velocity. This ensures that the Quadcopter stays close to z position when it reaches there.

Question 2: Discuss your agent briefly, using the following questions as a guide:

- What learning 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.

Answer: I combined the DDPG agent with task into a single function. This allowed me to make an exhaustive list of all the hyperparameters that i can control. In general, for a simple take off (X=0,Y=0,Z=100). The following strategy works really well.

1. Reduce the action zize to 1. i.e. control only 1 rotor and apply the same speed to other rotors.
2. Change the reward function to be optimized for the simple take off.

The rest of the hyperparameters are default. I used actor-crituque DDPG model that was provided and i did try to modify it an exhaustive way but none of the changes resulted in any significant improvement for me. A complete list of all the hyperparameters that i discoverd and approaches that i tried is shown below.

In [13]:
```
from IPython.display import HTML, display
pd.set_option('display.max_colwidth', -1)
Table = pd.read_csv('Quadcopter.csv')
Table = Table.to_html(index = False)
display(HTML(str(Table)))
```

Category	Parameter	Parameter Effects	My Discovery
Task	Action Size	Dimension of each action. i.e. Rotors to control.	Reducing the number of rotors to control makes the agent more efficient.
Task	Action Low	Minimum value of each action dimension. i.e. Lowest speed allowed of any rotor	I assumed that this parameter will be really useful but Its only the minimum limit of rotor speed.
Task	Action High	Maximum value of each action dimension. i.e. Highest speed allowed of any rotor	I assumed that this parameter will be really useful but Its only the maximum limit of rotor speed.
Task	Reward Function	Feedback signal to train the quadcopter agent.	This is one of the key parameters to control.
Task	Runtime	Runtime of each run.	Increasing the runtime makes the episodes longer.
Task	Initial Position	Initial position of the quadcopter in (x,y,z) dimensions and the Euler angles.	Changing the initial position makes the problem harder.
Task	Initial Angle Velocites	Initial radians/second for each of the three Euler angles.	Changing Initial euler angles makes the problem harder.
Task	Initial Velocities	Initial velocity of the quadcopter in (x,y,z) dimensions.	Changing initial velocites makes the problem harder.
Task	Action Repeat	Default Set at 3. Number of times an action is repeated.	I wasn't able to figure out how this parameter effects the problem.
Task	State Size	Action repeat * Multiplied by 6?. Dimension of each state.	I also couldn't figure out what this parameter does.
Agent	DDPG Actor: Neural Network	Can update the network itself by different layer sizes, activations, add batch normalization, regularizers or additional loss function(s),	Adding more layers increases the complexity of the model without much improvement.
Agent	DDPG Critic: Neural Network	Can update the network itself by different layer sizes, activations, add batch normalization, regularizers or additional loss function(s),	Adding more layers increases the complexity of the model without much improvement.
Agent	DDPG Replay Buffer: Buffer Size	Maximum size of buffer	Increasing the buffer size does not necessarily result in better agent.
Agent	DDPG Replay Buffer: Batch Size	Size of each training batch	Did not result in any significant improvement.
Agent	DDPG: Gamma	Discount factor	Reducing Gamma makes the agent focus on short term goals.
Agent	DDPG: Tau	For soft update of target parameters / default 0.01; higher means use more local, less target	Increaseing Tau makes it longer for agent to converge to a solution.
Agent	DDPG: Noise Process Exploration Mu	Long-term mean of the process	I am not sure how changing mean affected the Agent.
Agent	DDPG: Noise Process Exploration Theta	Rate of reversion to mean	Determines how long the agent will keep exploring.
Agent	DDPG: Noise Process Exploration Sigma	Volatility of the Brownian motion	Determines how hard will it be for agent to filter noise effectively.
Agent	DDPG: Number of Episodes	Number of Episodes to Train per Each Agent	Increasing the episodes is worth it. Agent does seem to get stuck in local minima solutions.

Question 3: 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)

Answer: its very hard to train an agent to control the quadcopter for any general task. The solution i provided only worked because the task was simplified to a simple vertical take off and the reward function was designed to accomodate that one task. I also reduced the action input to one rotor for the final improvement.

In the end, the agent performs really well but this is due to problem being manipulated from a general task to a very specific task. From the above plots on final simulation, i can make the following points

- The agent learns to take action in the right direction (Plot 1).
- The agent moves toward the direction of z in the entire run time (Plot 2).
- The agent does not change direction or wonder off in any way (Plot 3).
- The agent maintains a constant velocity of z = 25 most of the runtime (Plot 4).
- The agent does not have any angular velocities (Plot 5).
- The agent maintains a near constant rotor speed Plot 6).

Question 4: 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, plotting, specifying the task, etc.)
- Did you find anything interesting in how the quadcopter or your agent behaved?

Answer: its hard to get started because its difficult to wrap your head around the problem. In the beginning, some simulations were ending in less than runtime and i couldn't figure out why that was happening. I kept debugging my code to see if i made a mistake but the issue was that quadcopter often crashes. What i found interesting is that sometime the agent will consider crashing as a good solution. To counter this, i think that adding a heavy penalty to reward function for hovering below 5 should do the trick but then again, depending on the given task the respective penalty might not be a good idea.

In general, i think i completed the rubric for the project but an ultimate solution with a general agent might be solved with following ideas. some possible solutions that i could think of are;

1. Switcher Program that switches between different agents and tasks to complete a complex task. This solution is akin to assembly line of car production. it is very difficult for a single worker to manufacture the car completely on his/her own. However, it is easier to divide the task of manufacturing a car into many smaller industrialized tasks that each non-skilled worker can do. The same logic will apply in this solution. it might be easier to train 4 agents in the following manner to accomplish a harder task

- Agent 1 to approach Z position.
- Agent 2 to approach Y position while maintaining the Z position achieved early.
- Agent 3 to approach X position while maintaining the Z and Y position achieved early.
- Agent 4 to maintain the targeted position and make final adjustments.

1. Reduce the 3d Target position (x,y,z) into a single value (w) in such a that way that mapping is 1-1 for each coordinate value.

$$F(x, y, z) = w$$

I am not sure if this is a possibility but there might be a one dimension representation of x,y,z coordinate that can be used to train our quadcopter agent. This is akin to applying principal components to reduce the dimensions of data and then apply a respective learning algorithm to do the predictions.

1. Convert the continous action space into discrete action space. This is going to make the problem more simple to solve and it will work relatively well in real-world.

My reason for thinking this was is as following

- Quadcopters generally have a range of 2,000 meters or so but we might be able to restrict that range to 100 meters.
- Continous space can be rounded to an approximate discrete coordinate space. i.e. (10.11,2.7,10.9) can be rounded to (10,3,11).

Both of these points tell me that we might be able to get away from using continous action space.