# Commercial aircraft control using Deep reinforcement learning

Xudong Liu Lingjia Meng Buqing Nie

January 11, 2019

**Abstract**

In recent years, autopilot systems have been becoming more and more popular in both scientific and engineering fields. In this project, we use the famous pilot simulation software FlightGear as environment, with the help of the state-of-th-art Deep reinforcement learning algorithms DQN and PPO2 to implement autopilot systems. The result is not satisfying but promising .

# Contents

# 1 Introduction and Motivation

In recent years, autopilot systems have been becoming more and more popular in both scientific and engineering fields. General goals for unmanned aerial vehicle(UAV) control can be divided into two levels: mission-level and self-attitude-level. Mission-level objectives such as navigation require the aerial vehicle to follow specific trajectories, while self-attitude-level objectives require the aerial vehicle to maintain its own stability coping with turbulence.

There have been different approaches proposed to deal with this problem. For example, classical control methods like PID controller have shown good performance in many stable environments. However, when it comes to unpredictable environments including unknown dynamics, PID seems not to work well. Another popular approach is reinforcement learning(RL) which is successful in solving many complex problems. And there are chances that a well-trained RL agent can perform even better than human experts.

Though tireless effort has been made on this problem and some indeed got exceptional performance, auto control for aerial vehicle is still an open field, especially for commercial airplanes for which it's difficult to take experiments. In this project, we're going to use reinforcement techniques to design a controller for a commercial airplane to accomplish goals in both mission-level and self-attitude-level. And as said above, since it's hard to carry out experiments in reality, we will test our controller in a flight simulator called FlightGear.

# 2 Background

## 2.1 FlightGear

FlightGear is an open-source flight simulator in which users can choose a wide variety of models of airplanes for their own purpose such as pilot training, research and academic use or just personal interest. FlightGear has a relatively complete framework containing different flight dynamic models, sky model and world scenery data base. Since its source code have been kept open, it becomes possible for researchers to make extensibility and test different algorithms for autopilot in it.

In our project, we are using the model of Cessna 550/551 Citation.

## 2.2 Problem Statement

Since FlightGear has implemented flight dynamic models, what we need to do is to let our controller communicate with FlightGear by receiving data from and send data to it. The property tree provides complete information of the airplane, such as the attitude of airplane(including euler angles), velocities and acceleration along different axes and orientaions, and the location of airplane(including altitude, latitude, longitude). Through UDP, we will

receive these data at a certain rate. And Since a trajectory-following problem for airplane can be divided into different processes where airplane heads towards a certain waypoint, we should also specify the location of a target point(determined by latitude and longitude). In order to generalize the representation of a state, we use the relative position of airplane regarding the target point. So a state $S$ is defined related to the values of attitude, velocities, acceleration and relative position to target of airplane.

What we need to control(actions) are values of aileron(controls airplane's rolling), elevator(controls airplane's climbing and diving), rudder(controls airplane's turning), and two throttles(controls airplane's accelerating) of airplane. The definition of rewards for training can be found in attachment files.

# 3 Implementation Details

## 3.1 Summary

In this section, we want to introduce our python interface for flightgear.

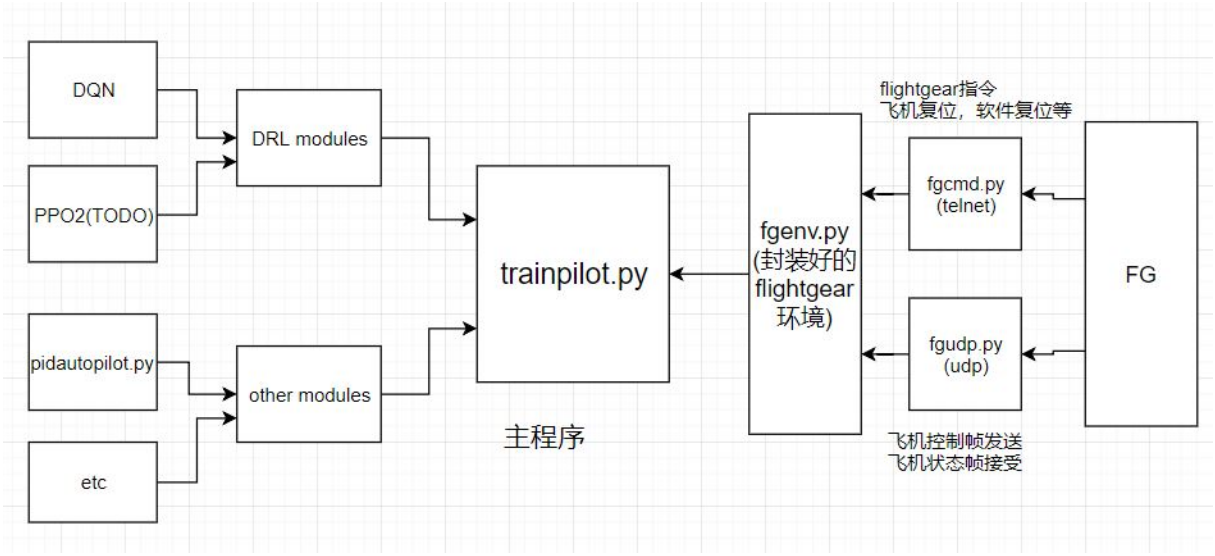


**Figure 1:** Training Framework

As show in the picture above, this is our whole framework.

1. *trainpilot.py* This is the main program, we intialize fgenv and Deep reinforcement learning modules and do our traning process here.

2. *DRLmodel* we setup our Deep reinforcement learning model here, DQN is available now.

3. *fgmodule* this is a module communicating with flightgear. it offers an python interface for flightgear.*fgudp.py* is mainly for receiving flight state from flightgear and send control frame ot it.*fgcmd.py*is a module that implements remote control for flightgear, In fact, it is an command interface for flightgear. *fgenv.py* is an packaged environment for flightgear like python open source lib *gym*, it contains *fgudp* and *fgcmd* module.

4. *other modules* some extra modules are stored in there, an PID autopilot program is available now.

## 3.2 Property Tree

In the property tree, we can see all the properties in simlutaion. the properties are stored in a tree structure. we need to use this tree in many place, so we have an breif introducation here.

FlightGear allows users to access a very large number of internal state(for example the flight dynamics models, or the weather simulation) variables via numerous internal and external access mechanisms. These state variables are organized into a convenient hierarchal property tree. With property tree, its possible to remotely control FlightGear from an external script or create own extensibility by editing some configuration files(The property tree maps very nicely to XML files). Property tree is just like an interface helping us keeping track of most of the internal information of the system, including the state of airplane.

Fisrt, we have an embedded web server in flightgear, to start it we only need to add Command line parameters in below.

- -httpd=5500

After start FlightGear, we can fire up a web browser and open up the following url: http://localhost:5480/.We can now browse the entire FG property tree "live" as the sim is running. We can set control inputs so we could literally fly the airplane from your web browser, although it's maybe not the most convenient interface for doing that.

Besides there, we also can get Property tree form path: *FG_ROOT/data/Docs/README.properties*. But there There are many important properties can't be found here.Finding from Web browser is a much more better choice.

## 3.3 UDP connection

**Communication Protocol** FlightGear has a few network access protocols. These are started via a command line option. More details are available in path:*FG_ROOT/data/Docs/README.IO*


**Figure 2:** Property Tree in Web Browser

FlightGear provides a generic protocol as a UDP packet to send out data (like the state of airplane, weather simulation) which can be read by our external program using standard socket communication. This same basic mechanism can also be used to send control commands to FlightGear, which makes it possible for us to test our UAV controller.

These configuration files can be written in XML layout with specification of variables we want access to. Available parameters and details of writting format can be found on FlightGear's official site. After finishing configuration files, we need to set corresponding serial ports to build communication.

This is an Generic protocol provided by FlightGear, since we can create an xml file specifying exactly what data values we wish to send or receive across your IO channel. the command line option are list below:

- -generic=socket,in,10,127.0.0.1,5701,udp,udp_input
- -generic=socket,out,10,127.0.0.1,5700,udp,udp_output

Then add file *udp_input.xml*, *udp_output.xml* to folder*FG_ROOT/data/Protocol*.
About how to define *udp.xml* you can read *FG_ROOT/data/Docs/README.protocol* and http://wiki.flightgear.org/Generic_protocol.
About the meaning of parameters you can read *FG_ROOT/data/Docs/README.IO*

After finishing the jobs listed above, we establish a udp sever in FlightGear successfully.More importantly we write an UDP cilent in python instead of C for this project. It works very well and more flexible.

## 3.4 telnet communication

### 3.4.1 brief introduction for telnet

Telnet is a protocol used on the Internet or local area network to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. It can be used to send command to target computer. In Python, a telnet lib is given for telnet communication.

For FLIGHTGEAR, we have introduced the function of the udp communication, which is used to receive plane information and send control frame. However, udp isn't able to control the FLIGHTGEAR main program, like *reset*, *re-position* command must be send to the main program by telnet.

### 3.4.2 how to use telnet

In python, a telnet library *telnetlib* is supported, which is used to set up the telnet communication client efficiently. The following are the work in this area.

1. To communicate between the python program and FlightGear, some Command line arguments are needed for flightgear: The first row: set up the 5501 port for telnet communication.

   The second row: allow other programs to send *nasal* commands, the flightgear will ignore the *nasal* commands from telnet without this setting.

2. Use the python telnetlib to set up the telnet client, the commands are aent to flightgeaar by *Telnet.write(self, cmd.encode("ascii"))*, and *(i,match,resp) = Telnet.expect(self, self.prompt, self.timeout)* is used to get the return result of the telnet.

   Up to now, a remote shell has set up to control the FlightGear main program.

3. FlightGear provide many classes of commands, here are some of them:

   - built-in commands: The built-in commands like *reset* to restart the simulation after the plane is crashed, and *re-position* to set the plane to the original position after crashed on the ground.

     Usage: run built-in-command

   - nasal commands: Nasal is FlightGear's built-in scripting language. The nasal commands can implement more advanced functions, like *save load* to save and load scenarios and all the built-in commands. Actually, a lot of built-in commands are implemented by nasal commands. But the communication speed is not satisfactory compared to built-in commands.

     Usage: nasal nasal-code ##EOF##

- set command: set the value of the nodes in property-tree the given value. This command can be used to switch perspective, pause the simulation, let the simulation crash.For example: set /sim/model/autostart true can switch perspective.

  Usage: set var value

### 3.4.3 The hard road to telnet

We read the source code of the flightgear, and the following are what we got:

1. The tennet connection:

   Location: /flightgear/scripts/python/FlightGear.py

   The FlightGear.py is a demo of the telnet connection script in python2, which is a big breakthrough of our work. Then we read the Telnet Usage in flightger Wiki, the telnet connection is based on these works.

2. The built-in commands file:

   Location1: /src/Main/fg_commands.cxx The following are some built-in commands find in this file.

```
/**
 * Table of built-in commands.
 *
 * New commands do not have to be added here; any module in the application
 * can add a new command using globals->get_commands()->addCommand(...).
 */
static struct {
  const char * name;
  SGCommandMgr::command_t command;
} built_ins [] = {
    { "null", do_null },
    { "nasal", do_nasal },
    { "pause", do_pause },
    { "load", do_load },
    { "save", do_save },
    { "save-tape", do_save_tape },
    { "load-tape", do_load_tape },
    { "view-cycle", do_view_cycle },
    { "property-toggle", do_property_toggle },
    { "property-assign", do_property_assign },
    { "property-adjust", do_property_adjust },
    { "property-multiply", do_property_multiply },
    { "property-swap", do_property_swap },
```

```
    { "property-scale", do_property_scale },
    { "property-cycle", do_property_cycle },
    { "property-randomize", do_property_randomize },
    { "property-interpolate", do_property_interpolate },
    { "data-logging-commit", do_data_logging_commit },
    { "log-level", do_log_level },
    { "replay", do_replay },
    { "loadxml", do_load_xml_to_proptree},
    { "savexml", do_save_xml_from_proptree },
    { "xmlhttprequest", do_load_xml_from_url },
    { "profiler-start", do_profiler_start },
    { "profiler-stop",  do_profiler_stop },
};
```

The file also include their implementation in C.

Location2: /docs-mini/README.properties This file include some properties of the plane, which can be used for *set* command. More propertices can be gotten by the control webpage.

Location3:/docs-mini/README.commands This file also contains some commands useful.

## 3.5 DQN

The state of the plane is hard to describe. Because lake of the expert knowledge, all the attributes of the plane we can get are all used to describe the state of the plane. Because some attributes of DQN, the state space and action space are discredited.

The following are some attributes in the state space: aileron, elevator, rudder, airspeed-kt, heading-deg etc. The value of these attributes are continuous, which are discredited to use in the DQN.

In the takeoff stage, some expert knowledge tell us the only thing we shoulg control is rudder. It controls the direction of the plane. Other atributes are fixed, like the accelerator should be in max state(expert knowledge). So the only thing controlled by the DQN is the rudder. The actions DQN can output are $action = [0, 1, 2]$

- $0 \rightarrow$ left
- $1 \rightarrow$ no action
- $2 \rightarrow$ right

$$rudder+ = 0.02 * (action - 1)$$

This design makes the action space become very small, which is good to the convergence. From another perspective, the range of the action is also very small (only 0.1), which makes the takeoff more smoothly.

The Q-network we used is a simple network with one hidden layer. Actually, we were planned to use more complex Q-network, but the convergence will become more slow if the Q-network becomes more complex. The training of the plane is very slow because of the restriction of FLightGear's running speed, so a simpler Q-network is better for training.

## 3.6 PID

The distinguishing feature of the PID controller is the ability to use the three control terms of proportional, integral and derivative influence on the controller output to apply accurate and optimal control

1. Term P is proportional to the current value of the SPPV error e(t).
2. Term I accounts for past values of the SPPV error and integrates them over time to produce the I term.
3. Term D is a best estimate of the future trend of the SPPV error, based on its current rate of change.

$$u(t) = K_{\mathrm{p}} e(t) + K_{\mathrm{i}} \int_0^t e(t')\,dt' + K_{\mathrm{d}} \frac{de(t)}{dt}$$

In this project, we use $de(t)$ rather than $\frac{de(t)}{dt}$, since $dt$ is hard to measure.

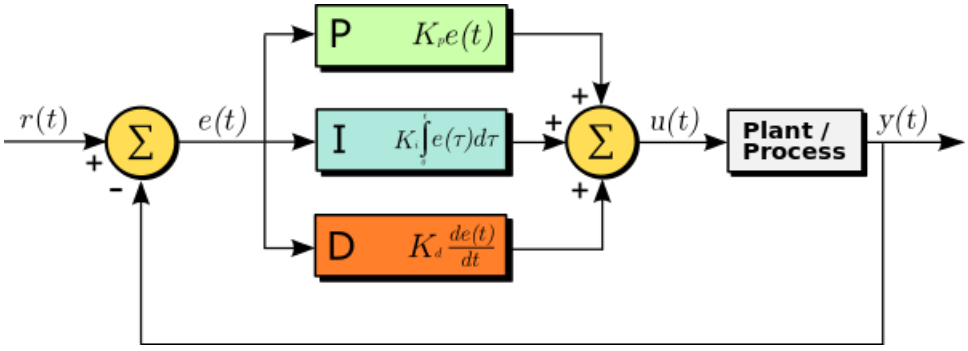


**Figure 3:** PID

## 3.7 Reward

We design different reward function for different state. Details in Appendix and Source code folder *fgmodule*.

# 4 Experiments

## 4.1 python interface inner delay

The udp communication has some delay which is harmful for the control of the plane. since we use python udp socket communication and three thread to do data processing, there are some delay inner python runing. But after experiment, we find that python interface inner delay is vey small, it's less than 0.1 ms,which is far less than plane state frame receiving rate 0.1s, which is satisfactory in this project.

## 4.2 pid running test

The pid control was tested in this project, whose performance is satisfactory. The plane controlled by pid accelerated and maintained direction on the runway, then stably climb and maintain the body posture after leaving the ground. Overall, the performance of pid is okay.

## 4.3 dqn training test

We tested DQN in the takeoff stage. The performance is not quite satisfactory. The speed of the convergence is very slow, which is the main problem we meet. Firstly, the plane always turn left/right immediately. Gradually, the plane can speed up for quite a long time, however, the plane still can't take off smoothly.

Besides, the flightgear has some unexpected bug. After sending some reset command by the telnet, the flightgear may crash down unexpecctedly. If we only use reposition command, after some epoches, the simulation will be cranky. Thus, the reset and reposition commands must use alternately.

# 5 Source code

## 5.1 *fgenv.py*

This is part code of *fgenv.py*,it was packaged like an gym class. Details about returns are as below:

1. *ob(object):* an environment-specific object representing your observation of the environment.

2. *reward(float):* amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increaseyour total reward.

3. *episode_over(bool):* whether it's time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being True indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)

4. *info(dict):* diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). However, official evaluations of your agent are not allowed to use this for learning.

```
class fgenv:
    def step(self,action):
        state_dict, _ = self.fgudp.send_controlframe(action)
        # todo : add function of data normlize
        ob = self.state_dict2ob(state_dict)
        # reward need to modify
        reward = self.calreward(state_dict)
        # finished with flightgear property node /sim/crashed
        episode_over = self.judge_over(state_dict)
        info = None
        return  ob, reward, episode_over, info
```

### 5.2 *DQN*

this is our simple dqn agent, it has three layers.

```
class DQN():
    # DQN agent
    def create_Q_network(self):
         # network weights
        W1 = self.weight_variable([self.state_dim, 20])
        b1 = self.bias_variable([20])
        W2 = self.weight_variable([20, self.action_dim])
        b2 = self.bias_variable([self.action_dim])
        # input layer
        self.state_input = tf.placeholder("float", [None, self.state_dim])
        # hidden layers
        h_layer = tf.nn.relu(tf.matmul(self.state_input, W1) + b1)
        # Q Value layer
        self.Q_value = tf.matmul(h_layer, W2) + b2
```

## 6 Discussion and Conclusions

### 6.1 Problem of pid controller

After test, the pid controller is able to control the plane in the takeoff stage and cruise stage. However, in the cruise stage, the plane will crash unexpectedly. We think this problem may be caused by the little disturbance in the FlightGear environment, and the pid controller can't approach the disturbance correctly.

## 6.2 Problem of DQN

The Q-network in the DQN we now used is a simple network with one hidden layer, we think the more complex one will cause slow convergence with better result.

Furthermore, the action space of the DQN is discrete, which means we must do discretization, which will makes the action space very big. Big action space is harmful to the convergence of DQN.

## 6.3 Future work

The DQN should be changed to PPO2, whose action space can be continuous. The completed pid controller will be used for data collection and to help the training of the PPO2 model.

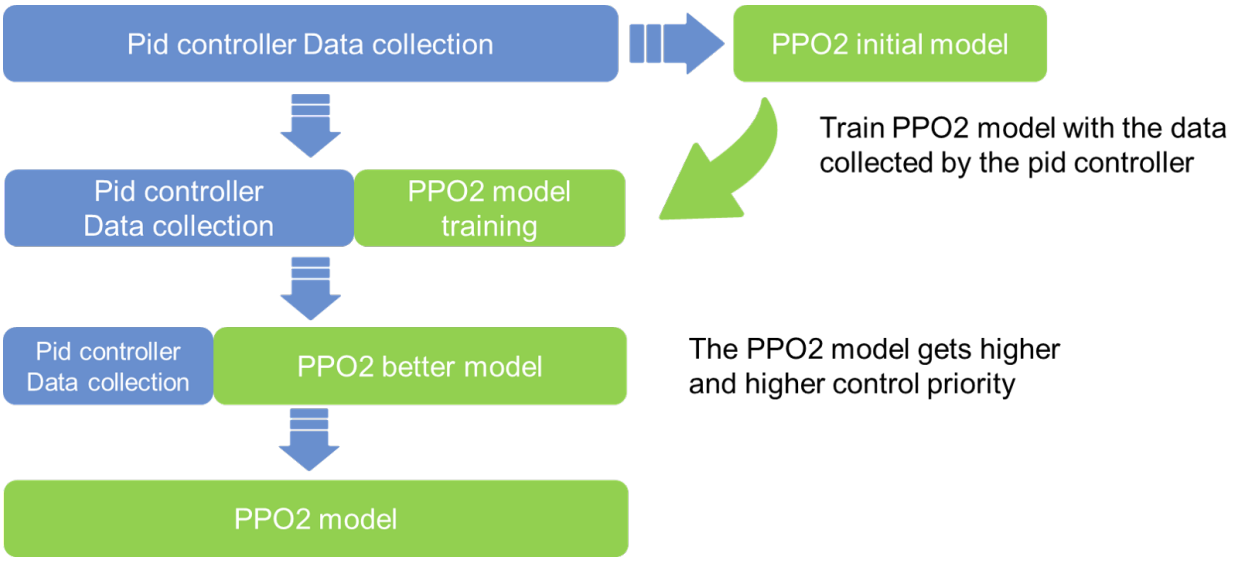


**Figure 4:** The future model

# Acknowledgments

Thanks to the teacher for the wonderful courses in this semester, thanks to the assistant for the guidance of the project. Thanks to the persistence of the members in the FlightGear group!

# Appendix

- reward.doc : This file is the reward functions in the different stages of the autopilot systems.
- reward.py : This file is the python implementation of the reward functions
- property.pdf : This file contains all the properties of the plane and specific explanations we collected.

# References


[1] http://wiki.flightgear.org/Telnet_usage

[2] https://blog.openai.com/openai-baselines-ppo/

[3] https://github.com/openai/baselines

[4] https://github.com/hill-a/stable-baselines

[5] https://morvanzhou.github.io/tutorials/machine-learning/reinforcement-learning/4-1-A-DQN/

[6] https://zhuanlan.zhihu.com/p/21477488

[7] DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning, XUE BIN PENG and GLEN BERSETH, KANGKANG YIN, MICHIEL VAN DE PANNE

[8] Proximal Policy Optimization Algorithms, John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov

[9] Control of a Quadrotor with Reinforcement Learning, Jemin Hwangbo, Inkyu Sa, Roland Siegwart, Marco Hutter

[10] Adaptive Neural Network Control of AUVs With Control Input Nonlinearities Using Reinforcement Learning, Rongxin Cui, Chenguang Yang, Yang Li, Sanjay Sharma

[11] Reinforcement Learning for UAV Attitude Control, William Koch, Renato Mancuso, Richard West, Azer Bestavros