Reinforcement Learning Based Autopilot

Muhammad Rizwan Malik
rizwanm@usc.edu

Muhammad Oneeb Ul Haq Khan
mkhan250@usc.edu

Martin Huang
hhuang04@usc.edu

Krishnateja Gunda
kgunda@usc.edu

Rengapriya Aravindan
raravind@usc.edu

October 19, 2021

Engineering Design Document	Version 1.0
Reinforcement Learning Based Autopilot	17th October 2021

Revision History

Date	Version	Description	Author(s)
10/17/21	1.0	Mid-Term Submission	Project Group

Contents

1 Introduction

1.1 Purpose

This Engineering Design Document (EDD) provides an overview of the Reinforcement Learning (RL) Based Autopilot agent implemented on the X-Plane 11 flight simulator as a course project for *CS527 - Applied Machine Learning for Games* at the **University of Southern California** for the Fall 2021 term.

This EDD provides the necessary background context as well as a discussion on the existing setup (i.e. existing code base, the environment and the tools used), as well as an in-depth look at steps taken by the project team to design multiple RL agents capable of autonomous flight on a flight simulator (X-Plane 11). This document will also touch upon the design decisions made by the team and the reasoning behind them, as well as reflect on the successes and failures.

1.2 Goal

The goal of this project is to be able to train Autopilot RL agents using neural networks and Policy Gradient algorithms. The proposed autopilot should be able to perform various aircraft maneuvers such as descent/ascent and left/right turns as required by a given flight plan. During the course of this project, we will explore different RL strategies in order to determine the most suitable approach for our particular use-case vis-á-vis a comparative analysis.

1.3 Definitions, Acronyms and Abbreviations

- **RL** - Reinforcement Learning
- **DDPG** - Deep Deterministic Policy Gradient
- **PPO** - Proximal Policy Optimization
- **EDD** - Engineering Design Document
- **XP11** - X-Plane 11
- **XPC** - X-Plane Connect
- **IAS** - Intelligent Autopilot System

2 Prior Research

Automatic Flight Control Systems are generally limited in terms of what they can control and perform. They are reserved for non-emergency related systems and have to be monitored frequently. They are not suited for sudden changes, in case of emergency and will start malfunctioning if an unexpected situation or any deviation from the normal were to happen. The paper [1] proposed an alternate solution using an Intelligent Autopilot System (IAS).

IAS face difficulty in being able to handle every scenario that is possible swiftly and accurately. This is where Artificial Neural Networks come in with their capacity to handle large amount of data that is dynamic. In the proposed system, it is learning by imitation, a concept that can also be applied to machines. In their 10 attempts to fly

the aeroplane autonomously,it was noticed that the system (Supervised learning using Artificial Neural Networks) developed was able to do well in calm as well as stormy weather conditions.

3 Background

3.1 X-Plane 11

For our RL Agents' environment, we will be using a flight simulator software. There are many popular flight simulator software that many actual pilots and aviation enthusiasts use. However for our particular case, we will be using X-Plane 11. X-Plane 11 allows us many advantages over general flight simulators. X-Plane 11 allows users the capability to read and write data to the simulator, which is discussed at length in 3.1.1. For this reason, X-Plane 11 is a popular choice for many Aerospace researchers and engineers.

Another reason XP11 is preferred by researchers is because instead of calculating aerodynamic forces such as lift and drag by using empirical data in pre-defined lookup tables, as is done by general flight simulator software, XP11 solves aerodynamic equations in real time. This allows XP11 to keep the simulation as close to reality as possible. It uses blade element theory, a surface (e.g. wing) may be made up of many sections (typically 1 to 4), and each section is further divided into as many as 10 separate subsections. After that, the lift and drag of each section are calculated, and the resulting effect is applied to the whole aircraft. When this process is applied to each component, the simulated aircraft will fly similar to its real-life counterpart.

Figure 1: X-Plane 11 Flight Simulator used for this project

3.1.1 Datarefs and Data Reference types

X-Plane provides data parameters which can be read from or written to the simulator through UDP sockets. These Data Parameters are called *data refs*. The X-Plane API allows the sharing of data with X-Plane as well as other plugins. The most common use of the X-Plane APIs is to read data from X-Plane and change the values within X-Plane.

Datarefs can be considered as variables or an object that represent a value. All of the communication with the X-Plane takes place by reading and writing data references. When we read a data reference, code inside X-Plane provides the value of the dataref.

If in the future the layout of the internal variables in X-Plane changes, the same data references may be used to access the new variables.

Data Reference types:
Each data reference can be read in one or more formats. Data references are defined via distinct bits in an enumeration; we can add them together to form sets of datatypes.

Examples of Datarefs:

Name	Type	Writable	Units	Description
sim / flightmodel / position / groundspeed	float	n	meters/sec	The ground speed of the aircraft
sim / cockpit2 / gauges / indicators / altitude_ft_pilot	float	n	feet	Indicated height, MSL, in feet

3.2 X-Plane Connect (XPC)

There are different options to communicate with X-Plane. We can use our own utility function and socket for communication between XP11 and the agent. Or use existing plugins such as the NASA X-Plane Connect plugin which provides different function calls to send/read current control information inside the simulator without the use of sockets. XPC just references the Data references. The XPC Toolbox is an open source research tool used to interact with XP11. XPC allows users to control aircraft and receive state information from aircraft simulated in X-Plane using functions written in C, C++, Java, MATLAB, or Python in real time over the network.

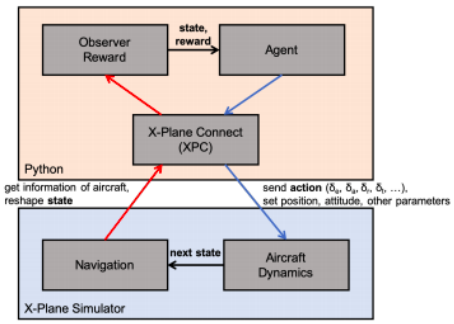


Figure 2: Connection of X-Plane with Python via XPC.

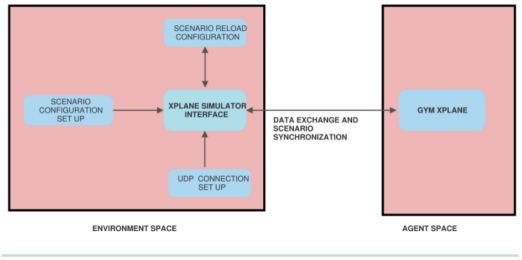


Figure 3: Agent-Environment Interaction Flow.

3.3 OpenAI Gym

The existing OpenAI Gym setup is a perfect addition to the suite of tools at our disposal. In the basic model of Reinforcement Learning, the agent interacts with the environment in discrete time steps. In each time step, the agent receives state and reward from the environment. Then the agent decides the action based on it's policy and the environment outputs next state after executing the action. After executing the specific action, XPC receives it and sends it to X-Plane using the prepared function. After sending the action, X-Plane simulator calculates the new state from the flight dynamics. XPC has a receiving function where we can get the information of the aircraft like position, velocity and other settings. Using OpenAI Gym allows us to

introduce new RL algorithms in a truly *plug-and-play* fashion, which is infact its raison d'être. The Gym set up allows us to reset the environment, take a step, calculate reward, as well as get the current observation space. We discuss this implementation in greater detail in 4.2.

4 Methodology

4.1 Overview

The solution to any game based RL training problem can be divided into following parts:

1. The first step is to configure the environment using which the agent will be trained. In our case this includes setting up the communication link between the Python script and the simulator. All the parameters will be passed through this link, i.e. XPC.
2. One of the most important tasks in solving a problem using RL is defining the reward function. It becomes even more important when dealing with a close-to-reality and complicated environment like flight simulation. Reward function is the only way an agent knows what is it supposed to learn and also plays a key role in determining how quickly it learns it.
3. Artificial neural networks are good function approximators whenever trying to learn complicated non-linear models.
4. Training the network based on policy gradient approach. Our network will basically output different actions that can be taken for a given observation and the reward function will evaluate whether that particular action in the given state took us closer to the target or not and that will be utilized to learn a policy for reward maximization.
5. Once the network is trained, it will be tested and results will be compared against the current data.

4.2 Simulation Environment

Reinforcement Learning problems require the agent to sense the environment, choose an action after evaluating the current policy for the sensed state, perform that action and sense the new state of the environment. Open AI provides an abstraction layer to perform these tasks in the form of Open AI Gym framework for several games and other simplified environments. However, Gym does not have any environments for X-Plane 11 so we had to write our own environment following the Gym API guidelines to some extent. The environment consists of following parts:

4.2.1 space_definition.py

This file extends the definition of Gym environment spaces to our problem. The action space is a 4 item box space and the observation space is an 8 item box space. The action space consists of following parameters:

Action Space Parameter	Type	Range
Latitudinal Stick	Box	[-1,1]
Longitudinal Stick	Box	[-1,1]
Rudder Pedal	Box	[-1,1]
Throttle	Box	[-1,1]

The choice of observation parameters is moved mainly by the relevance of different physical parameters to the actual aerodynamics model. Therefore the observation space consists of following parameters:

Observation Space Parameter	Type	Range
Indicated Airspeed	Box	[0,inf]
Vertical Speed	Box	[-inf,inf]
Altitude above MSL	Box	[0,inf]
Pitch	Box	[-180,180]
Roll	Box	[-180, 180]
Heading	Box	[-360, 360]
Angle of Attack	Box	[-180, 180]
Sideslip Angle	Box	[-180,180]

4.2.2 parameters.py:

This is a utility file which contains dictionaries of datarefs which will be used by the environment to get or set parameter values in the X-Plane 11 simulator. The datarefs that are used in our observation and action space are as follows:

DataRef	**Data Type**	**Description**
sim / flightmodel / position / indicated_airspeed	float	Indicated airspeed of the aircraft
sim / flightmodel / position / vh_ind	float	Indicated vertical speed of the aircraft
sim / flightmodel / position / elevation	int	Elevation of the aicraft above MSL
sim / flightmodel / position / theta	float	Pitch of the aircraft
sim / flightmodel / position / phi	float	Roll angle of the aircraft
sim / flightmodel / position / true_psi	float	True heading of the aircraft relative to true geographic north
sim / flightmodel / position / alpha	float	Angle of attach of the aircraft relative to the wind
sim / flightmodel / position / beta	float	Sideslip angle of the relative wind

4.2.3 xpc.py:

This is NASA Xplane Connect file which is used to as an interface to communicate with the simulator. Some of the functions which we used are:

- getDREF(dref): gets a particular dataref from the simulator.
- getDREFs(dref_list): gets a list of datarefs from the simulator.
- getCTRL(): gets the current position of controls in a list [latitudinal_stick, logitudinal_stick, rudder_pedal, throttle, gear, flaps, speedbrakes]
- sendCTRL(CTRL_list): writes the control parameters in the simulator.
- pauseSim(bool): pauses the simulation. This does not actually pause the whole simulator, it simply stops the physics engine of X-Plane 11. This is used whenever we need to perform the learn operation for the agent and then resuming the simulation after that.

4.2.4 xplane_envBase.py:

This is the main file of the environment which ties together the RL agent and the X-Plane 11 simulator. This file has some of the functions of a typical Open AI Gym environment file. The functions provided by the environment file include:

- connect(): This calls the X Plane Connect's connect() function and sets up a connection between the simulator and our environment.
- close(): This function disconnects the UDP connection between the simulator and the environment.
- step(actions): This function accepts a list of actions as the argument and passes those to the simulator and returns the [observation space, reward, done, info] to the agent after those actions are performed. This function makes use of following helper functions:
  - getObservationSpace(): This passes the datarefs dictionary defined in parameters.py file to x Plane Connect and fetches the results.
  - **getReward(state):** This function computes the reward for being the current state. The reward function is defined as:
    - reward = -20 for each timestep
    - reward = $+6000$ for being inside the target zone
    - reward = $-\sqrt{|current\_altitude - target\_altitude|}$
    - reward = -100000 in case the aircraft crashes
    - reward = -20000 if the episode and ends and the aircraft was not in the target zone
  - checkTerminalState(): This function sets the done state flag to indicate the end of the episode.
- reset(): This function is called by the agent at the start of each episode and it resets the environment and returns the initial state. A point which is pertinent to mention here is that the whole training of the model is done using a situation file of X-Plane, which is essentially a configuration file. However, X-Plane 11 and the

NASA X-Plane Connect do not provide any commandref API to reload a situation file through Python script. Therefore, to solve this problem we used another open source library written in Lua programming language called FlyWithLua. FlyWithLua is loaded as a plugin when X-Plane is run and it keeps checking the specified parameters regularly. Whenever the plane crashes or 2000 steps are completed, our Python script sets a flag in the X-Plane and upon reading that flag our Lua script reloads the situation file.

4.3 Reinforcement Learning

4.3.1 REINFORCE

REINFORCE is a Monte-Carlo variant of policy gradients (Monte-Carlo: taking random samples). The agent collects a trajectory of one episode using its current policy, and uses it to update the policy parameter. Since one full trajectory must be completed to construct a sample space, REINFORCE is updated in an off-policy way. So, the flow of the algorithm is:

Vanilla REINFORCE

for *Episode = 1 ... 1000* **do**
  Input: Initial observation
  for *Step = 1 ... 2000* **do**
    Perform a trajectory roll-out using the current policy
    Sample action values from the given normal distributions with μ and σ values
    Calculate reward
    Store action taken and reward received
  end for
  Calculate discounted cumulative future reward at each step
  Compute the loss for each step using the product of log probability of the action taken and the discounted cumulative reward
  Compute policy gradient and update the network parameters
end for

Implementation: Our implementation of the REINFORCE algorithm was done using the following agent file.

1. **REINFORCE_Agent.py:** The agent is divided into sections, the *Policy Network* itself and the *Training loop*.

   - **Policy Network** comprises of 4 layers, an input layer, 2 hidden layers and an output later. The input layer is of size 8, followed by the hidden layers of size 256 with a ReLU activation function each. The output layer is also of size 8, however we have 2 different sets of outputs μ and σ. The μ values are outputted as they are, however for the σ values are passed through an ELU activation function.
   - **Training Loop** does an off-policy roll-out for each episode, and then updates the network parameters by calculating the loss for each step and then computing the gradients once the episode has ended.

4.3.2 Deep Deterministic Policy Gradient (DDPG)

DDPG is a reinforcement learning technique that combines both Q-learning and Policy gradients. DDPG is an actor-critic technique wherein the actor is a policy network that takes the state as input and outputs the exact action (continuous), instead of a probability distribution over actions. The critic is a Q-value network that takes in state and action as input and outputs the Q-value. DDPG is an "off"-policy method. DDPG is used in the continuous action setting and the "deterministic" in DDPG refers to the fact that the actor computes the action directly instead of a probability distribution over actions. DDPG is used in a continuous action setting and is an improvement over the vanilla actor-critic.

DDPG

Input: initial policy parameter θ, Q function parameters ϕ, empty replay buffer D
Set target parameters equals to main parameters $\theta_{targ} = \theta$, $\phi_{targ} = \phi$
for *Episode = 1 ... 1000* **do**
  Observe state s and select action $a = \text{clip}(\mu_\theta + \epsilon, -1, 1)$, where ϵ is normal distribution
  Execute a in the environment
  Observe next state $s\prime$, reward r, and done signal d to indicate whether $s\prime$ is terminal
  Store s, a, r, $s\prime$, d in replay buffer D
  If $s\prime$ is terminal, reset environment state
  If it's time to update then **for** *Step = 1 ... 2000* **do**
    Randomly sample a batch of transitions, $B = (s, a, r, s\prime, d)$ from D
    Compute targets: $y(r, s\prime, d) = r + \gamma(1-d)Q_{\phi targ}(s\prime, \mu_{\theta targ}(s\prime))$
    Update Q function by one step of gradient descent using: $\bigtriangledown_\phi \frac{1}{|B|} \sum_{(s,a,r,s\prime,d)} (Q_\phi(s,a) - y(r,s\prime,d))^2$
    Update policy by one step of gradient ascent using: $\bigtriangledown_\phi \frac{1}{|B|} \sum_s Q_\phi(s, \mu_\theta(s))$
    Update target networks with: $\phi_{targ} = \rho\phi_{targ} + (1-\rho)\phi$, $\theta_{targ} = \rho\theta_{targ} + (1-\rho)\theta$
  end for
end for

Implementation: Our implementation of the DDPG algorithm was done using the following agent file.

1. **DDPG_agent.py:** contains the agent function, actor network, critic network, noise function, and replay buffer function for the DDPG network.
2. **DDPG.py**: contains the X-plane environment setup, training loop and plot utility

4.3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization belongs to a family of Policy Gradient based Reinforcement Learning algorithms called Actor-Critic methods. Actor-Critic methods are better suited for problems dealing with continuous action spaces. In Actor-Critic methods there are two policy networks being trained simultaneously.

One network learns the actual policy to behave optimally given a state, this network is the Actor. The second network learns the value function of the underlying MDP, this network is called Critic because it criticizes how the Actor network evaluates the rewards of a state while updating its parameters.

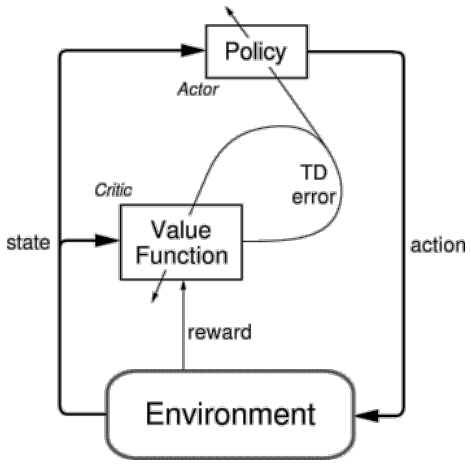


Figure 4: Actor-Critic Methods

To understand the main difference between vanilla REINFORCE algorithm and PPO, we need to look into the optimization objectives of both of these methods. The loss function for vanilla policy gradient is given as:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[log\pi_\theta(a_t|s_t)\hat{A}_t]$$

In the above equation $L^{PG}(\theta)$ is the policy loss and it is equal to the expected value of taking action a_t in state s_t. The expected value is weighted by the estimated advantage function $\hat{A}_t$, which in case of vanilla policy gradient techniques like REINFORCE is simply the discounted rewards. However, in the paper Trust Region Policy Optimization (Schulman et al, 2015) the authors introduced a concept of *Trust Regions* to limit the policy gradient step so it does not move too much away from the original policy, causing overly large updates that often ruin the policy altogether.
For this, they define $r(\theta)$ as the probability ratio between the action under the current policy and the action under the previous policy.

$$r_t\theta = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Given a sequence of sampled actions and states, $r(\theta)$ will be greater than one if the particular action is more probable for the current policy than it is for the old policy. It will be between 0 and 1 when the action is less probable for our current policy. Since our action space is continuous and we sample the actions from 4 uncorrelated normal distributions. Therefore, instead of directly dividing the probabilities of actions, we take exponentials of probability density functions.
The loss function is defined using the probability ratio as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

Here, the expectation is being computer over a minimum of two terms: normal policy gradient objective and clipped policy gradient objective. The second term plays a key

role where the objective value is clamped between 1-epsilon and 1+epsilon, epsilon being the hyperparameter. The paper uses the $\epsilon = 0.2$.
Furthermore, because of the min operation, this objective behaves differently when the advantage estimate is positive or negative.
One aspect of PPO which makes it more suitable for our problem is its sample efficiency. It makes use of a memory replay buffer to store the sample actions and then trains the model for several epochs over that data before discarding it, unlike REINFORCE algorithm where an experience trajectory is used only once to train.
The algorithm from the paper is as follows:

PPO with clipped objective

Input: initial policy parameters θ_o, clipping threshold ϵ
for *k = 0, 1, 2, ...* **do**
Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}^{\pi_k}$ using any advantage estimation algorithm
Compute policy update

$$\theta_{k+1} = argmax_\theta \mathcal{L}^{CLIP}_{\theta_k}(\theta)$$

by taking K steps of minibatch SGD via Adam where

$$\mathcal{L}^{CLIP}_{\theta_k}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

end for

Implementation: We followed the standard implementation of a PPO agent. Our PPO code consists of following files:

1. **PPO_main.py:** This file sets training and batch_sizing parameters for our agent. These parameters include the number of episodes, batch_size, mini_batch_size & epochs. It initializes the environment as well as the PPO agent and performs the iteration, num_episodes times. Inside the main *for* loop, there is a *while* loop which runs during each episode until *done*.

2. **PPO_Agent.py:** This file contains three classes namely *PPO_Memory, ActorNetwork, CriticNetwork*.

   - *PPO_Memory* initializes all the data structures (in this case python lists) to store the agent actions and experiences through a partial trajectory.
   - *ActorNetwork* initializes the actor network which consists of 4 layers of neurons including 2 hidden layers of each 256 neuron. The activation functions between the hidden layers are *ReLU()*. At the output layer for *mu* there is no activation whereas the *sigma* values are obtained after $ELU(x) + 1.0001$ function. These *mu* and *sigma* are then used to construct the Gaussian distribution.

   - *CriticNetwork*: This network takes in the environment state at the input layer and gives out the value function approximation at the output layer. This network has 1 hidden layer of 256 neurons with *Relu* activation.

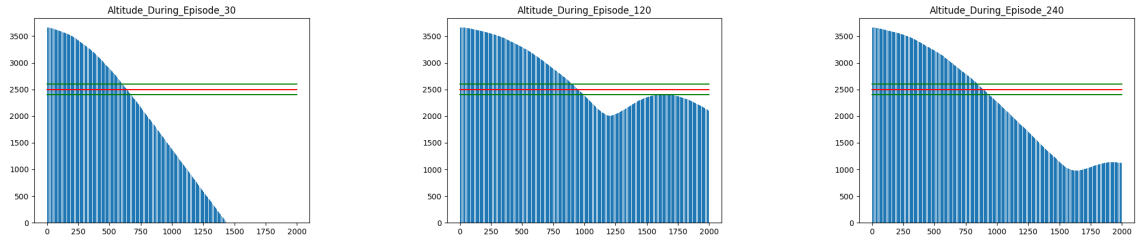
the workhorse of this file is the *learn()* function. This function is called by the *PPO_main.py* file whenever the *PPO_Memory* buffer is full. This function makes use of some utility functions to divide the experience relay memory into mini_batches. For each batch, this function passes each state through *ActorNetwork* and *Critic Network* then for each batch calculates the advantage estimate. Then using the new and old *log_probabilities* it computes the probability ratio and *clipped_loss*. This loss is then backpropagated through the respective networks and the parameters are updated.

5 Results and Analysis

5.1 REINFORCE

REINFORCE is widely known as a *vanilla* implementation of Policy Gradient algorithms. It is not an algorithm of choice for many RL applications because it is very slow to converge. We see this replicated behaviour in our implementation as well. Since both our observation and action spaces are extremely large, this behaviour of REINFORCE results in our agent being frustratingly slow to learn. We have been able to train the agent for a maximum of 2000 episodes, however even that has proven to be insufficient. There have been some promising results and we have run several different iterations of REINFORCE, however none has proven to be satisfactory. The conclusion drawn from this is that our focus needs to shift towards more efficient algorithms like DDPG and PPO, which should learn the objective much faster.

In Figure we see some promising trends. Our agent is tasked to learn to fly at an altitude of 2500m. The agent needs to descend from 3660m, its starting point, descend to 2500m and maintain that altitude. Initially, the agent crashes the plane consistently, as is observed in Figure 6a, however it eventually learns to not crash and then eventually also starts trying to come back to 2500m after it has descended past it, as observed in Figure 6b. However, instead of improving upon this, we see in Figure 6c that the agent continues to descend past the target altitude. This trend continues for several hundred more episodes, after which the training for this run was terminated.



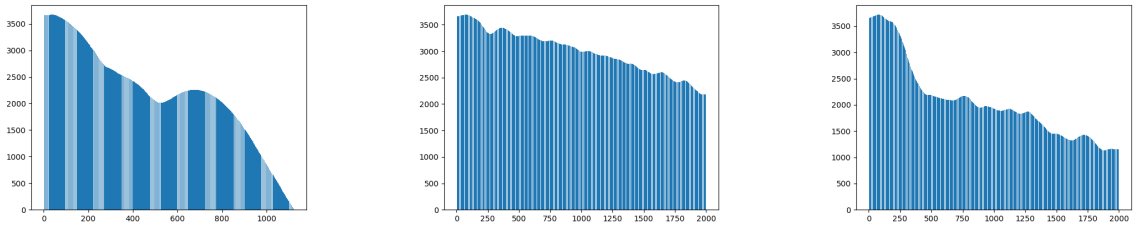
(a) Episode 30 (b) Episode 120 (c) Episode 240

Figure 5: Altitude of the aircraft at each time step for a given episode

5.2 Deep Deterministic Policy Gradient

In Figure 6, we can see how the agent is learned based on the reward function. After running 121 episodes, the agent could well know that there is huge reward during 2500 ft, and try to level up to obtain the reward. After 220 episode, the agent knows to keep the altitude near 2500 ft at the end of the max actions. Then, we have noticed that

there is interesting behavior learned by the agent at the end of 1000 episodes, since we have absolute altitude penalty and reward, which is not based on previous state, then the agent knows to dive down at the first in order to gain less penalty. Then it tries to maintain the altitude, which is not optimal. Therefore, the agent sometimes give up to control the plane and let it dive down till crash to obtain least penalty. This problem could be solved if we further change the reward function to include the previous state or increase the episode for let the agent fully experience the environment.



(a) Episode 121 (b) Episode 220 (c) Episode 998

Figure 6: Altitude of the aircraft at each time step for a given episode

5.3 Proximal Policy Optimization

Results for PPO algorithm have been inconclusive in our case.

1. The first case ran for 500 episodes with following hyperparameters:

   - memory_buffer = 1000
   - mini_batch_Size = 100
   - n_epoch = 5
   - learning_rate (α) = 0.0003
   - epsilon (ϵ) = 0.2

   Average scores for the last 100 episodes did not show a lot of variation and the descent trajectory that the aircraft followed did not change either over the course of 500 episodes.

(a) Episode 10 (b) Episode 490

Figure 7: Altitude of the aircraft at each time step

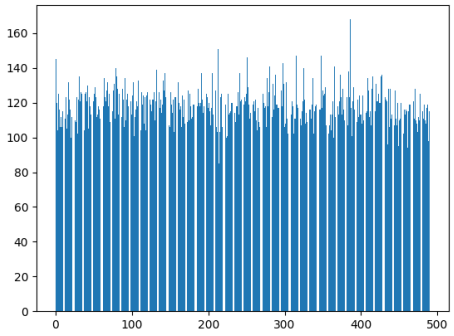


Figure 8: Number of Successful steps in each episode

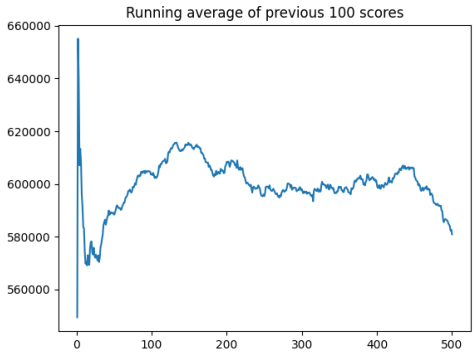


Figure 9: Avg score of last 100 episodes

6 Future Work

For the second half of the semester, we will be focusing heavily on PPO and DDPG implementations, making the necessary adjustments to ensure that target objectives are being met. Currently, the goal has been to maintain altitude, once this is achieved, we will expand the goal objective to include maintaining heading as required as well. Once this objective is achieved, we will shift our focus so that our agent learns to achieve any target altitude and heading. This will allow us to piece together entire flight plans, which the agent should be able to follow. Another focus of the team will be on running XP11 on a cloud instance, more for logistical ease than for performance boosting.

References

[1] Haitham Baomar and Peter J Bentley. "An Intelligent Autopilot System that learns piloting skills from human pilots by imitation". In: *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE. 2016, pp. 1023–1031.

[2] Jean de Becdelievre et al. "Autonomous Aerobatic Airplane Control with Reinforcement Learning". In: (2016).

[3] Yenn Berthelot. *AI learns to fly — Airplane simulation and Reinforcement Learning*. [Online; accessed 26-April-2020]. 2020.

[4] Yenn Berthelot. *AI learns to fly — Create your custom Reinforcement Learning environment and train your agent*. [Online; accessed 25-August-2020]. 2020.

[5] *Deep Deterministic Policy Gradient*. `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`.

[6] William Good. *FlyWithLua for X-Plane 11*. 2020.

[7] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3 (1992), pp. 229–256.

[8] *X-Plane Connect*. `https://github.com/nasa/XPlaneConnect`.

[9] *X-Plane Datarefs*. `https://developer.x-plane.com/datarefs/`.

[10] Takeshi Tsuchiya Yuji Shimizu. "Construction of Deep Reinforcement Learning Environment for Aircraft using X-Plane". In: *Machine learning* 8.3 (2020), pp. 112–119.