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

December 8, 2021

Engineering Design Document	Version 2.0
Reinforcement Learning Based Autopilot	8th December 2021

Revision History

Date	Version	Description	Author(s)
10/17/21	1.0	Mid-Term Submission	Project Group
12/08/21	2.0	Final 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), including 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 have explored 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
- **SAC** - Soft Actor Critic
- **EDD** - Engineering Design Document
- **XP11** - X-Plane 11
- **XPC** - X-Plane Connect
- **IAS** - Intelligent Autopilot System

2 Prior Research

Past research in the field of autopilot system has been focused primarily on control system theory. Classic and modern autopilots are based mostly on Proportional Integrated Derivative (PID) controllers or Finite-State automation. However, very limited research has been done in exploring self-learning or experiential learning autopilots. In the past, one of the limiting factors has been computational intractability of the close to infinite state-space of the real life flight dynamics model and another has been limited research into efficient algorithms to handle large and continuous state spaces. Recent efforts towards the development of Intelligent Autopilot System:

1. **IAS based on imitation learning:** This implementation is based on [1], where a training dataset was first collected of a human pilot flying an aircraft on a simulator. This dataset was then used to train an Artificial Neural Network (ANN) based model. This report acts as a proof of concept that experiential learning can be used to train neural networks to control an aircraft.

2. **A Domain-Knowledge-Aided Deep Reinforcement Learning Approach for Flight Control:** This implementation is based on [10] where they leverage domain knowledge to improve learning efficiency and generalisability of aircraft control. This implementation employs a Markovian decision process with a proper reward function, allowing reinforcement learning theory to be used. Domain knowledge is also used to define the reward function by molding reference inputs in consideration of crucial control objectives and using the shaped reference inputs in the reward function.

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.

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

Table 1: Examples of Datarefs

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.

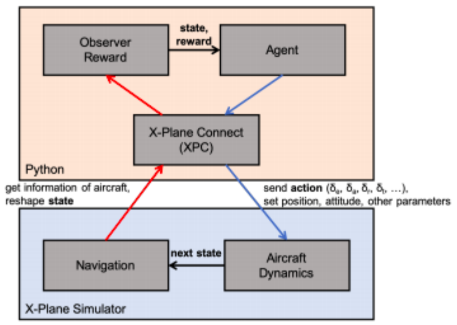

Observer Reward
state, reward
Agent
X-Plane Connect (XPC)
Python
get information of aircraft, reshape **state**
send **action** (δ_e, δ_a, δ_r, δ_t, ...), set position, attitude, other parameters
Navigation
next state
Aircraft Dynamics
X-Plane Simulator

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

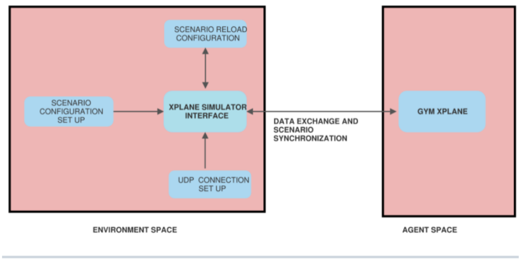

SCENARIO RELOAD CONFIGURATION
SCENARIO CONFIGURATION SET UP
XPLANE SIMULATOR INTERFACE
UDP CONNECTION SET UP
DATA EXCHANGE AND SCENARIO SYNCHRONIZATION
GYM XPLANE
ENVIRONMENT SPACE
AGENT SPACE

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 the following steps:

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

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

Table 2: Action Space Parameters

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 tha parameters in Table 2.
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 parameters in Table 3

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]

Table 3: Observation Space Parameters

However, for soft actor-critic agent we limit the observation space to only the first 3 datarefs in Table 3

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:

4.2.3 xpc.py:

This is the NASA Xplane Connect file which is used as an interface to facilitate communication between the simulator and our agent (python script). 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.

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

Table 4: Observation Space Datarefs

- 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 $\mu$
         and $\sigma$ 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 an off-policy Actor-Critic Policy Gradient method that utilizes Q-Learning. Actor-Critic methods are well-suited for problems dealing with continuous action spaces [12]. Since we calculate the actions directly instead of their probability distributions, DDPG is a deterministic method, hence its name Deep *Deterministic* Policy Gradient. Because of its deterministic nature, there is not much room for exploration. In order to cater to this, the authors of DDPG [11] introduce *Ornstein-Uhlenbeck* (OU) noise. OU Noise has the desired property of approaching its mean (μ) as $t \to \infty$. In DDPG, $\mu = 0$, so that as learning continues the noise becomes closer to zero, and the exploratory behaviour of the algorithm diminishes.

In this method 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, i.e. its Q Value. This second network is called the Critic because it criticizes how the Actor network evaluates the rewards of a state while updating its parameters. In order to calculate the loss for the Critic Network, we utilize 2 more networks, i.e. the Target Actor and Target Critic networks. We utilize a Mean Squared Error (MSE) loss function for the Critic Network to show roughly how closely Q_ϕ is to satisfying the Bellman equation:

$$L(\phi, D) = \underset{(s,a,r,s',d)\sim D}{E}[(Q_\phi(s,a) - \\ (r + \gamma(1-d)\underset{a'}{max} Q_\phi(s', a')))^2]$$

Target Actor and Target Critic networks provide us with the target value:

$$r + \gamma(1-d) max_{a'} Q_\phi(s', a')$$

Given all of this, DDPG achieves its objectives by minimizing the MSE loss with stochastic gradient descent between the target value and the output of the Critic network.

In DDPG, we save the state, action taken, reward, the next state, denoted by state', and the terminal flag at every step within the episode in a buffer. After every step, we then train our networks over a randomly sampled batch from this buffer. In our implementation the batch size is 5000. The state and action values are used to generate the critic value via the Critic network. The state' values are used to generate the target actions using the Target Actor network. The Target Critic network then uses the state' and target actions to generate the target critic values. The Critic network then uses the MSE of the critic value and target value, as specified in the previous equations as its loss value and tries to minimize it. The Actor network uses $-Q$ obtained from the

Critic network as its loss value. We then use a soft update approach to update the Target Actor and Target Critic networks, using a factor τ.

The Target Critic $\theta^{Q^{'}}$ is updated as follows:

$$\theta^{Q^{'}} = \tau\theta^{Q} + (1-\tau)\theta^{Q^{'}}$$

The Target Actor $\theta^{\mu^{'}}$ is updated as follows:

$$\theta^{\mu^{'}} = \tau\theta^{\mu} + (1-\tau)\theta^{\mu^{'}}$$

DDPG is used in a continuous action setting and is an improvement over the vanilla actor-critic. So the DDPG algorithm is:

DDPG

```
Input: initial policy parameter $\theta$, Q function parameters $\phi$, empty replay buffer $D$
Set target parameters equals to main parameters $\theta_{targ} = \theta$, $\phi_{targ} = \phi$
while True do
    Observe state $s$ and select action $a$ = clip($\mu_\theta + \epsilon$, -1, 1), where $\epsilon$ 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:
        $\nabla_\phi \frac{1}{|B|}\sum_{(s,a,r,s\prime,d)}(Q_\phi(s,a) - y(r, s\prime, d))^2$
        Update policy by one step of gradient ascent using:
        $\nabla_\phi \frac{1}{|B|}\sum_s Q_\phi(s, \mu_\theta(s))$
        Update target networks with: $\phi_{targ} = \rho\phi_{targ} + (1-\rho)\phi$,
        $\theta_{targ} = \rho\theta_{targ} + (1-\rho)\theta$
    end for
end while
```

Implementation: For all 4 networks being used in DDPG, we utilize a network architecture of $8 \times 400 \times 300 \times 4$. The input layer has 8 nodes for the observation space parameters, the 2 hidden layers have 400 and 300 nodes respectively and the output layer has 4 nodes, one for each action. The Actor and Target Actor networks use ReLU as the activation function in hidden layer 1 and 2, and a tanh activation function for the values ofμ, bounding them between -1 and +1. The Critic and Target Critic networks also use ReLU as their activation functions.

DDPG has a lot of different hyperparameters, and finding the optimal combination is a challenge of its own. For the Actor and Target Actor networks we started with a learning rate $eta = 2.5 \times 10^{-5}$, and for the Critic and Target Critic networks we started with a learning rate $eta = 2.5 \times 10^{-4}$. However, we soon determined, given our reward structure, these learning rates were too high and so they were changed to 2.5×10^{-7} and 2.5×10^{-6}, respectively. The buffer was set to 1×10^{6} steps, with a batch size of 5000. Since the training at every step, greatly reduced our training pace, we changed the frequency to every 20 steps. The soft update factor τ was set to 0.001.

1. **DDPG_Main.py**: the main training file, which interacts with the X-plane environment setup, contains the initial parameters of DDPG, the training loop and also makes calls to the helper functions in *utils.py* file for plotting and saving data.

2. **DDPG_Agent.py:** contains the *Agent*, *ActorNetwork*, *CriticNetwork*, *OUActionNoise*, and *ReplayBufffer* classes.

All of the functionality from the classes comes together in our *Agent* class. We initialize the Critic and Target Critic networks as instances of the *CriticNetwork* class, and the Actor and Target actor networks as instances of the *ActorNetwork* class, as well as initialize the noise and memory buffers as instances of the *ReplayBuffer* and *OUActionNoise* classes, respectively. When choosing an action we call our $\boldsymbol{choose}_{a}ctionfunction, whichsimplydoe$

4.3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization [8] 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.

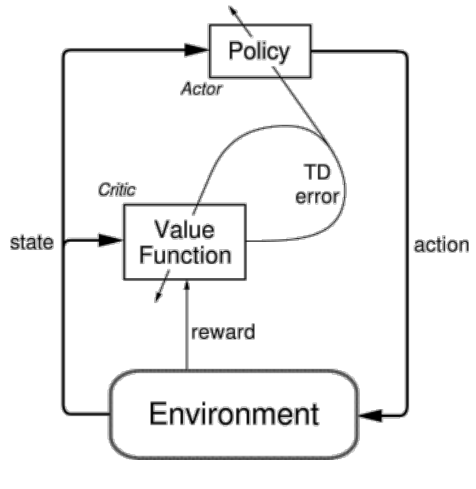


Figure 4: Actor-Critic Methods

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 [9] 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 computed 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 [8] uses the $\epsilon = 0.2$.
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 $\theta_o$, clipping threshold $\epsilon$
**for** $k = 0, 1, 2, \ldots$ **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 and implemented PPO in three different ways.

- PPO in Continuous Action Space

- PPO in Discrete Action Space using 4 networks

Implementation is similar for both the agents. However, there are differences in the way the actor network predicts the outputs which is discussed in item 2 of *PPO_Agent.py*. 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. The main training loop is inside this file which is run num_episodes times and during each episode, the inner *while* loop runs until the *done* flag is raised. This file also handles making calls to helper functions in *utils.py* file for plotting and saving data.

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()*. The output layer and activations for 3 implementations are as follows:
     - PPO in continuous action space: The output consists of 8 units representing μ and σ for the 4 actions. At the output layer for μ there is no activation function. However, the output units for σ are activated through $ELU(x) + 1.0001$ function to keep the σ positive. These outputs are then used to construct the Gaussian Distributions from which actions are sampled.
     - PPO in discrete action space with 4 networks: This network is designed in a way that the main agent consists of 4 actor networks. Each actor network is trained to predict one of the 4 actions. The action space is discretized to 11 equidistant points between [-1, 1]. Output layer is activated using Softmax activation function to output a discrete probability distribution function over the action space.
   - *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.

4.3.4 Soft Actor-Critic (SAC)

Model-free deep reinforcement learning (RL) algorithms have been shown to be able to solve a wide range of challenging decision making and control tasks. However, these methods typically suffer from two major challenges: very high sample complexity and brittle convergence properties, which demand an intense hyperparameter tuning. Both of these challenges severely limit the applicability of such methods to complex, real-world problems. Soft actor-critic, on the other hand, is an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning framework. In this framework, the actor aims to maximize expected reward while also maximizing entropy. That is, to succeed at the task while acting as randomly as possible. Soft Actor-Critic algorithm utilizes a parameterized state value function $V_\psi(s_t)$, soft Q-function $Q_\theta(s_t, a_t)$, and an action policy $\pi_\phi(a_t|s_t)$. The parameters of these networks are ψ, θ, and ϕ. For example, the value functions can be modeled as expressive neural networks where the network predicts a value function for the given state or state-action pair, and the policy as a Gaussian with mean and covariance given by neural networks. Based on the update equations of these parameters given in the paper [7], the algorithm can be written as:

Soft Actor-Critic

```
Initialize parameter vectors $\psi, \overline{\psi}, \theta, \phi$.
for each iteration do
  for each environment step do
    $a_t \sim \pi_\phi(a_t|s_t)$
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
  end for
  for each gradient step do
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i) for i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\overline{\psi} \leftarrow \tau\psi + (1-\tau)\overline{\psi}$
  end for
end for
```

The above algorithm makes use of two Q-functions to mitigate positive bias in the policy improvement step that is known to degrade performance of value based methods. In particular, the two Q-functions are parameterized, with parameters θ_i, and are trained independently to optimize $J_Q(\theta_i)$. Minimum of the two Q-values is used in the loss functions of value and policy gradients. The method alternates between collecting experience from the environment with the current policy and updating the function approximators using the stochastic gradients from batches sampled from a replay buffer. In our implementation, we take a single environment step followed by 5 gradient steps.

Implementation: We followed the standard implementation of SAC agent. Our SAC code consists of following files and their respective functionalities:

1. **buffer.py**: This file initializes the memory buffer, a cyclic buffer of size max_size for storing experiences. Experience is defined as a collection of $\{s_t, a_t, r_r, s_{t+1}, done_t\}$ at each time-step. It takes in parameters $max\_size, input\_shape, action\_shape$. It includes helper functions to sample random batches from the memory buffer and a function to store each experience.

2. **networks.py:** This file contains the actor, critic and value network architectures

and related functions.

- *Actor Network:* It is a fully connected neural network which takes state as input and produces μ and σ corresponding to the 4 actions at the output layer. The architecture consists of 3 hidden layers of 256 units in each layer. Input layer contains 3 units and output layer contains 8 units. Hidden layer outputs are activated using *ReLU* function.
- *Critic Network:* Critic network takes state and action as input and produces a single output representing the Q-value of the state action pair. It contains same hidden layers as the actor network. However, the input layer consists of 3+4 (representing 3 observation space and 4 action space) units and the output layers contains just 1 unit.
- *Value Network:* Value network takes state as input and produces a single value as output representing the value function of the state. It contains same hidden layers as the actor network. However, the input layer consists of 3 units (representing 3 observation space) and the output layers contains just 1 unit.

The function *sample_normal*() returns *action* for a given *state*. Whereas, the function *choose_test_action*() is used during testing and it returns the action value for a given state while test. It also includes functions for saving and loading the models.

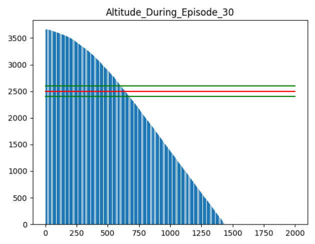
3. **sac_torch.py:** This file contains the *Agent* class and all of the associated functions. The *Agent* class' attributes include *actor*, *critic* & *value* networks as well as all the hyperparameters. The workhorse of *Agent* class is *learn*() function. This function performs all gradient descent and network parameters update after each environment step. During each gradient step, it calls the *memory_buffer* class function to draw a random batch from the replay buffer and performs network updates as described in the algorithm above.

5 Results and Analysis

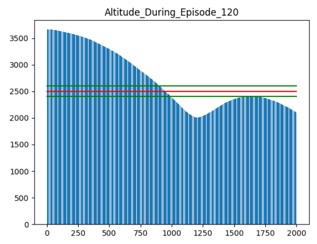
5.1 REINFORCE

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 behaviour replicated 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.

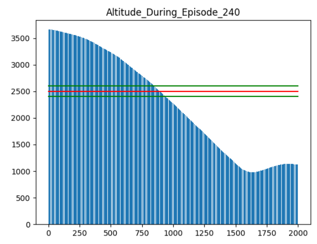
In Figure we see some promising trends. Our agent is tasked to learn to fly at an altitude of 2500m. The agent needs to descend from 3660m, its starting point, descend to 2500m and maintain that altitude. Initially, the agent crashes the plane consistently, as is observed in Figure 5a, however it eventually 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 5b. However, instead of improving upon this, we see in Figure 5c 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



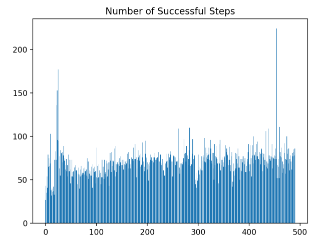
(c) Episode 240

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

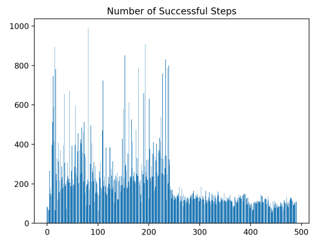
5.2 Deep Deterministic Policy Gradient

We analyze the performance of the agent using five different metrics. The first metric is the trajectory of the aircraft during each episode. The second is the difference between the aircraft's altitude at the end of the episode and the specified target altitude. The third is the number of successful steps the aircraft takes in one episode. The fourth is the running average of the reward accumulated during an episode for the last 100 episodes. The last metric are the loss values for the Critic and Actor networks.

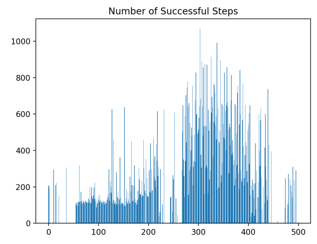
In our initial implementation, we observed a lot of promising results. In the first 500 episodes of training, the plane keeps crashing, as is shown in Figure 7a. However we see a marked improvement in the flight trajectory as we get closer to 2000 episodes of training, confirmed by Figure 7c.



(a) Episode 0 - 499



(b) Episode 500 - 999

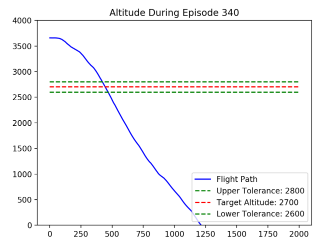


(c) Episode 1500 - 1999

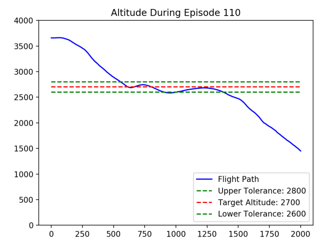
Figure 6: DDPG - Number of Successful Steps

As we continued to train the agent, upto episode 7780, we saw a deterioration in performance which the agent did not recover from. This, we learnt, was because our hyperparameters were not optimized. One of the hyperparameters that needed to be corrected was the learning rate, which was not adequately chosen for our reward structure.

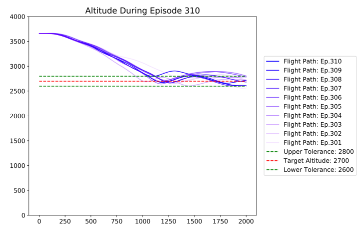
We restarted training our agent with a much smaller learning rate $\eta = 2.5 \times 10^{-7}$ for our Actor and Target Actor networks and $\eta = 2.5 \times 10^{-6}$ for our Critic and Target Critic networks. We also redesigned our episode structure so that the episode started with our aircraft within the target zone, and it would end once the aircraft flew outside



(a) Episode 0 - 499



(b) Episode 500 - 999

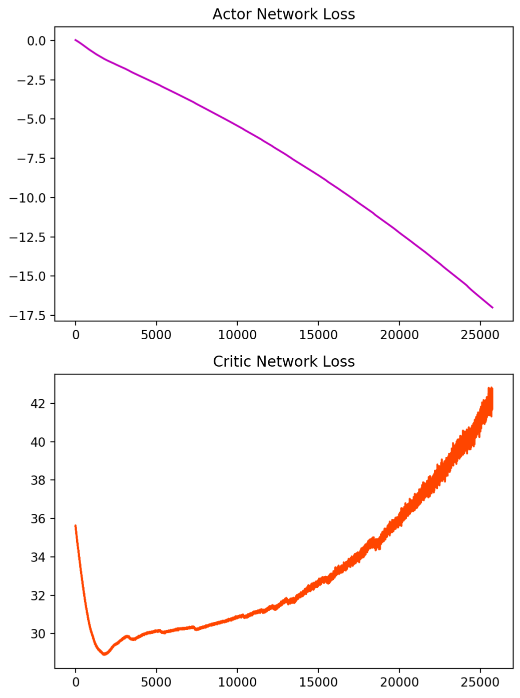


(c) Episode 1500 - 1999

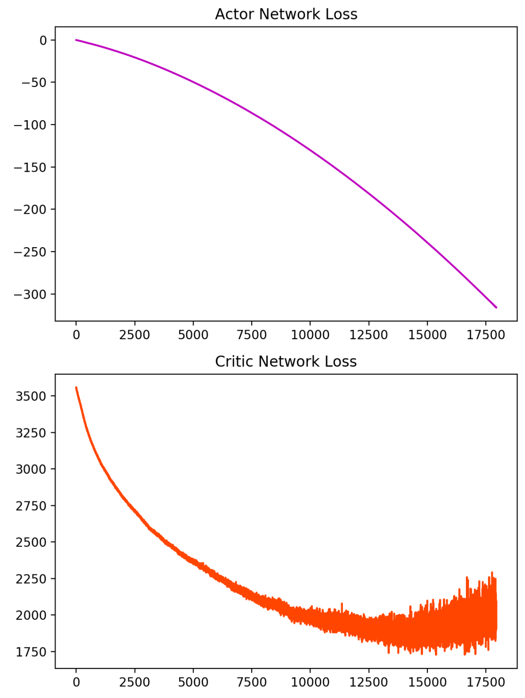
Figure 7: DDPG - Altitude During Episode

the target zone. The rationale behind this was to encourage the agent to fly within the target zone for longer.

Our focus now was to use the loss functions of the Actor and Critic networks as guides to improve the performance. By tweaking the learning rate we were able to see an improved performance in terms of loss, however the agent's performance still did not achieve satisfactory levels. We tried different values and combinations of learning rates, while also scaling the reward value. In doing so, we were able to improve the Critic network loss curve as can be seen from the change from Figure 8a to Figure 8b. However, due to time constraints, we were unable to perform an exhaustive grid search allowing us to determine optimal combination of the learning rates and other hyperparameters. Because of this reason, DDPG was unable to yield satisfactory results, although we were very optimistic about the direction the agent was headed in.



(a) Actor $\eta = 2.5 \times 10^{-7}$ Critic $\eta = 2.5 \times 10^{-6}$



(b) Actor $\eta = 2.5 \times 10^{-5}$ Critic $\eta = 2.5 \times 10^{-5}$

Figure 8: DDPG - Actor, Critic Network Losses

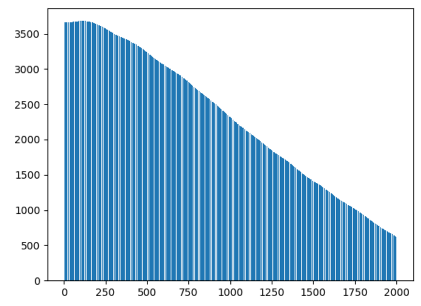
5.3 Proximal Policy Optimization

Results and analyses of different iterations of PPO algorithm are as follows

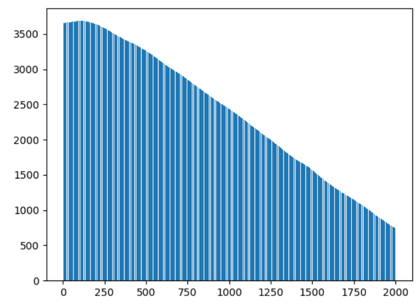
1. PPO in continuous action space ran for 500 episodes with following hyperparameters:

   - 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 as shown in Figure 11 and the descent trajectory that the aircraft followed did not change either over the course of 500 episodes which can be observer in Figure 9. We realised that the *memory_buffer* was very small and the number of epochs that we ran gradient descent for was also very small, hence the agent could not learn any patterns from the data.



(a) Episode 10



(b) Episode 490

Figure 9: PPO Continuous - Altitude of the aircraft at each time step

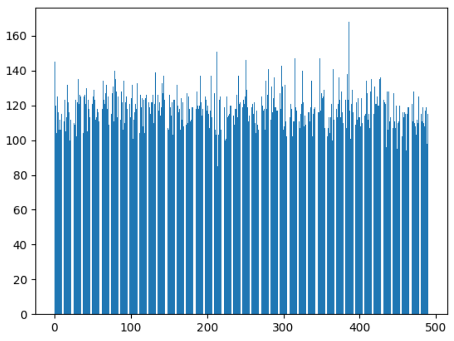


Figure 10: Number of Successful steps in each episode

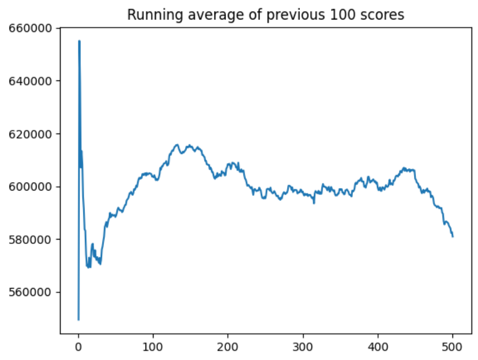
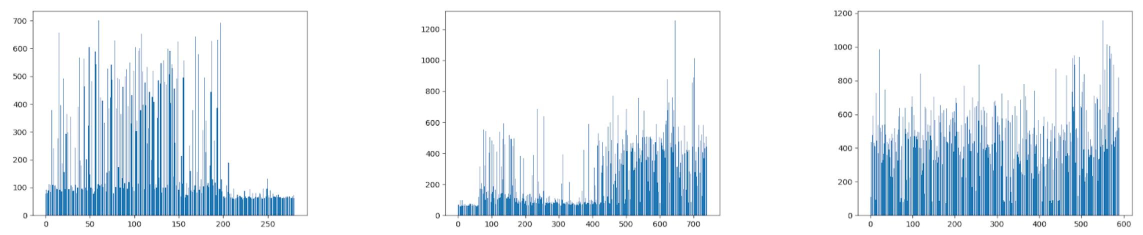


Figure 11: Avg score of last 100 episodes

2. PPO in discrete action space with 4 networks was run for close to 1700 episodes and it showed some good results with following hyperparameters:

- memory_buffer = 100000
- mini_batch_Size = 5000
- n_epoch = 50
- learning_rate (α) = 0.0003
- epsilon (ϵ) = 0.2

We can see in Figure 12 that the number of successful steps in each episode increased significantly as well as the trajectory of the aircraft moved closer to the target zone. During training the plane started crashing at one point, however, it recovered and improved performance as can be seen in the following pictures.



(a) Episode 0 - 280 (b) Episode 281 - 1030 (c) Episode 1031 - 1640

Figure 12: PPO Discrete - Successful Steps during each episode

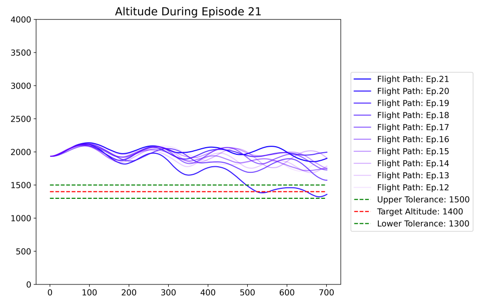
As discussed earlier, on-policy algorithms like PPO are very inefficient when it comes to sample complexity. Since, the whole memory buffer needs to be cleared after each gradient descent step. Therefore, it can be very difficult to perform hyperparameter tuning during a limited time. Hence, we decided to implement the state of the art Soft Actor-Critic algorithm instead of tuning PPO.

5.4 Soft Actor-Critic

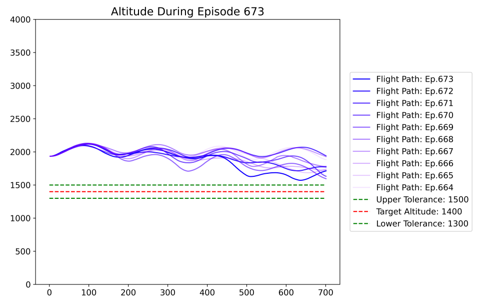
We performed 2 iterations of Soft Actor-Critic algorithm by changing some hyperparamters. The second iteration showed good results. The hyperparameters that were used during the first iteration are as follows:

- memory_buffer = 1000000
- batch_Size = 1024
- learning_rates (α, β) = 0.0003
- γ = 0.99
- target value update smoothing constant (τ) = 0.005
- reward scale = 1.0

During the training the agent seemed very conservative in exploration despite being formulated on entropy maximization framework. In the Figure 13 it can be observed that the trajectory of the aircraft during the episodes 663-673 is almost the same as that during the episodes 11-21. The scores also plateaued as can be seen in the Figure 15.



(a) Episode 11-21



(b) Episodes 663-673

Figure 13: SAC Reward Scale 1 - Altitude of the aircraft during an episode

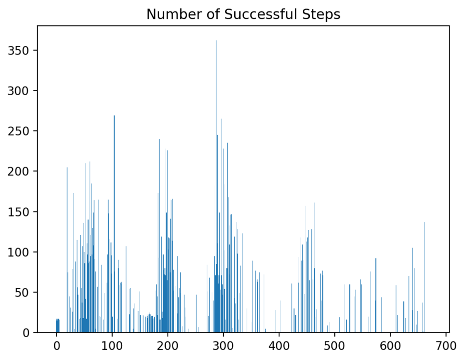


Figure 14: Number of Successful

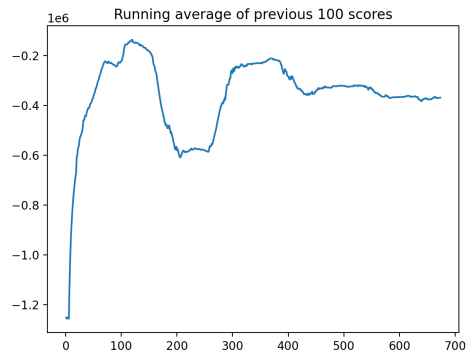
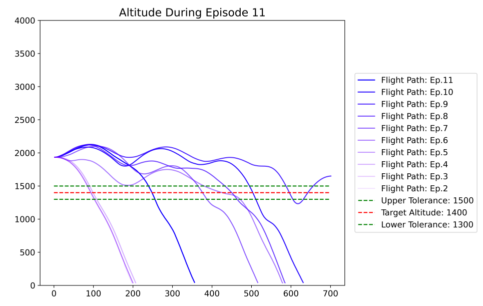
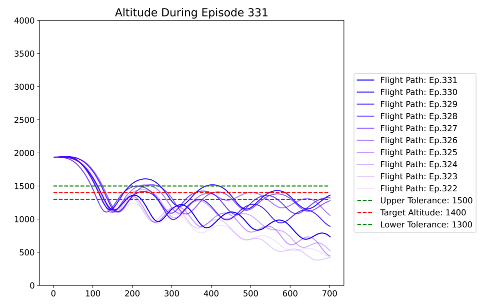


Figure 15: Avg score of last 100 episodes

SAC paper [7] mentions that the most important hyperparameter that can be tuned to improve performance of the agent and balance exploration vs exploitation is reward scale. The paper states that the range of reward scale is dependent on every environment, however, values between 10 - 100 result in better performance. We performed the second training iteration using reward scale of 10. We can observe from Figure 16 that the agent not only started exploring more, but it also recovered from crashed very quickly, since the reward was being weighted more while performing gradient descent of critic network.
Another thing that is worth noting in the above figures is oscillations in the trajectory of the aircraft. When the aircraft tries to recover from the dive, it pitches up, but during the pitch up motion if the agent predicts a severe action for aileron then the aircraft starts to bank as well. During the bank movement the it loses energy and therefore less engine power is available to recover and reorient. These oscillations eventually result in an overall loss of altitude.
Since we had very limited amount of time available, therefore, to make the model simpler for the agent we reduced the observation space to 3 parameters and also included a hard coded condition for aileron. This hard coded rule keeps the aircraft's bank angle within $\pm 10^{\circ}$.



(a) Episode 1-11



(b) Episodes 321-331

Figure 16: SAC Reward Scale 10 - Altitude of the aircraft during an episode

The model was trained and tested. Following three tests were performed with an initial altitude of 1836m. The agent was **successful** in all these tests.

- Descend to an altitude of 1600m
- Descend to an altitude of 1200m
- Climb to an altitude of 2000m

Results of the above tests were shown in the form of a video.

6 Future Work

Over the course of the project we worked on different approaches to develop an RL agent with the ability to learn to change altitude based on on-policy as well as off-policy algorithms. The experiments were a strong indicator towards the ability of off-policy algorithms to learn the models quickly.
Future efforts will include work on a more intuitive reward function representative of the experiment goals in a better way. We have seen that off-policy algorithms are very sensitive to hyperparameters. Therefore, significant improvement in results can be expected with careful hyperparameter tuning. During the semester we were constrained by time as well as compute power. Exploring cloud resources compatible with the simulator can provide an opportunity to not only improve the compute efficiency but also to run multiple experiments in parallel.
The results which we see after hard-coding aileron action to maintain the attitude during the experiment is indicative that curriculum learning can be a very interesting approach to explore in this context. It can be extended to include a wider spectrum of tasks. A curriculum based on increasingly complicated tasks assisted by relevant training data for learning by imitation can help the agent learn more complicated and realistic tasks including taking off, waypoint following and flying in close formation.

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] Tuomas Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[8] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[9] John Schulman et al. “Trust region policy optimization”. In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[10] Hyo-Sang Shin, Shaoming He, and Antonios Tsourdos. “A Domain-Knowledge-Aided Deep Reinforcement Learning Approach for Flight Control Design”. In: *arXiv preprint arXiv:1908.06884* (2019).

[11] David Silver et al. “Deterministic policy gradient algorithms”. In: *International conference on machine learning*. PMLR. 2014, pp. 387–395.

[12] Lillian Weng. *Policy Gradient Algorithms*. `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html`.

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

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

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

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