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

Abstract

State-of-the-art autopilot systems rely on control system theory, an approach which becomes extremely complex as demands from it increase. Our goal for this project is to develop a Reinforcement Learning (RL) agent, capable of learning basic flight maneuvers within a simulated environment (X-Plane 11). To this end, we use four RL algorithms, namely, REINFORCE, Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC).

***Keywords*—** Reinforcement Learning (RL), REINFORCE, Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG), Soft Actor-Critic (SAC)

1 Introduction

With the recent advances in the field of Reinforcement Learning and Deep Learning there has been an increased interest in solving problems which were considered intractable in the past. One of the common themes in the current research has been to develop models which can learn and perform tasks with close-to-human performance. The problem that is discussed in this paper relates to development of a intelligent autopilot system based on reinforcement learning.

Human pilots are trained to handle flight uncertainties or emergency situations such as severe weather conditions or system failure. In contrast, Automatic Flight Control Systems are highly limited in a way that they are only capable of performing minimal piloting tasks in non-emergency conditions. Strong turbulence, for example, can cause the autopilot to disengage or even attempt an undesired action which could jeopardise flight safety. Moreover, the current regulatory requirements by International Civil Aviation Organization (ICAO) require constant monitoring of the system and the flight status by the flight crew to react quickly to any undesired situation or emergencies. On the other hand, trying to cater for every possible uncertainty in the flight conditions by hard-coding it into the autopilot is not only impossible but futile as well.

This work aims to address these problems by proposing an RL based Intelligent Autopilot System (IAS) capable of performing different flight maneuvers. The proposed model will be trained and tested using X-Plane 11, an extremely popular and extensible flight simulator used by both aviation enthusiasts and professionals. Instead of having the agent learn by imitation as done in the past [1], our proposed IAS will learn by performing actions by following a policy and then improving the policy based on the rewards and penalties it receives from its environment.

This paper describes the overview of the X-Plane environment and the interaction of X-Plane via X-Plane Connect (XPC), as well as the implementation of RL algorithms and an in-depth discussion on the performance of each algorithm. This paper is structured as follows: **Section 2** covers the relevant prior research, **Section 3** deals with the specification and overview of the environment used for training the agent, **Section 4** concerns discussion of the algorithms which were implemented and **Section 5** presents results and analysis.

2 Related Work

Past research in the field of IAS 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 include:

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 [8] 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 Data and Environments

3.1 Overview of X-Plane 11

The environment in which the RL agents are trained is X-Plane 11 flight simulator. There are many other commercial flight simulators available, but we used X-Plane because it provides the capability to read and write live data from the simulator via UDP sockets. Also, unlike other flight simulators which calculate aerodynamic forces such as lift and drag by using empirical data in predefined lookup tables, X-Plane 11 solves aerodynamic equations in real time using blade-element theory. This allows X-Plane to simulate the flight conditions as close to reality as possible.

3.2 X-Plane Connect (XPC)

X-Plane 11 allows live data to be read and written in the form of data-references (datarefs) via UDP sockets. However, instead of focusing on writing sockets we used an existing plugin called

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

NASA X-Plane Connect (XPC). XPC was developed by the NASA Ames Research Center Diagnostics and Prognostics Group. It provides an abstraction layer between the environment and the flight simulator. Its API provides different function calls to send/receive current datarefs to and from the simulator without the need to dive into socket programming.

The XPC API, however, does not provide functionality to reload the X-Plane 11 flight configuration file i.e. the situation file, with file extension *.sit*. Therefore, another plugin called FlyWithLua is used to interact with the simulator and reload the situation file whenever a terminal state is reached.

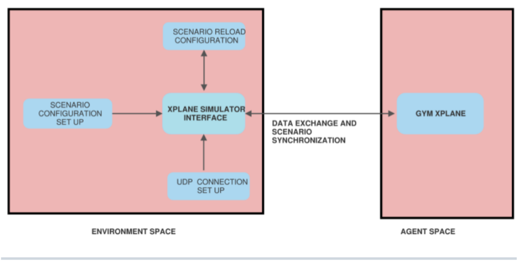

SCENARIO RELOAD CONFIGURATION
SCENARIO CONFIGURATION SET UP
XPLANE SIMULATOR INTERFACE
DATA EXCHANGE AND SCENARIO SYNCHRONIZATION
GYM XPLANE
UDP CONNECTION SET UP
ENVIRONMENT SPACE
AGENT SPACE

Figure 2: Agent-Environment InteractioFlow.

4 Methods

A reinforcement learning problem is generally modeled as a Markov Decision Process (MDP). An MDP is a process that takes an agent from one state to another, whereby the transition probabilities between different states depend only on the current state and the action the agent takes. For each state transition, the agent is given a reward r_t. The aim of the agent is to maximize the sum of future discounted rewards, also known as the gain

(G), at every timestep *t*:

$$G_t = \sum_{k=t}^{\infty} r_k \gamma^{k-t}$$

γ is the discount factor, which controls how rewards are weighed. A lower discount factor means that immediate rewards are preferred, while a discount factor close to 1 should be used for environments in which actions have long-lasting consequences.

A generalized solution of any game based RL training problem can be divided into the 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 it is 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. Therefore, all the algorithms discussed in this paper are implemented using multilayer perceptrons with non-linear activation functions.

4. The algorithms discussed in this paper are based on policy gradient approach wherein the agent attempts to learn the actual policy which outputs actions, instead of learning the value functions. The network will 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.

4.1 Simulation Environment

We modeled the problem as an episodic Markov Decision Process (MDP) in which the environment is reset once the agent steps into a terminal state. Such 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 until the episode terminates. 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 interfaced our own environment following the Gym API guidelines.

4.1.1 Observation Space

The observation space for the problem includes following flight parameters: (i) indicated airspeed, (ii) vertical velocity, (iii) relative altitude from the target altitude, (iv) pitch angle θ, (v) roll angle ϕ, (vi) true heading ψ, (vii) angle of attack α, (viii) side-slip angle β. The observation was scaled in an attempt to keep the input values between $\{0, 1.0\}$. In order to scale the observation a linear transformation was applied on each parameter of the observation space to bring the values inside $\{0.0, 1.0\}$

4.1.2 Action Space

The actions which are available to the agents are: (i) latitudinal stick for pitching motion, (ii) longitudinal stick for rolling motion, (iii) rudder pedals for yawing motion, (iv) throttle. The values for the first three actions are in the range [-1, 1] and for throttle it is [0, 1].

4.1.3 Rewards Function

The following reward function was implemented for the RL agent to be able to learn to maintain the target altitude:

- reward =

  $$-\sqrt{|current\_altitude - target\_altitude|}$$

  for every step when the aircraft is not in the target zone, which is defined as: $(target\_altitude \pm 100)$

- reward = $+6000$ for being inside the target zone

- reward = -100000 in case the aircraft crashes

- reward = -20000 if the episode and ends and the aircraft was not within the target zone

4.2 Reinforcement Learning Agents

Policy Gradient algorithms are a branch of RL algorithms which focus on optimizing the policy. Policy Gradient algorithms are well-suited for continuous spaces, since value-based algorithms become computationally intractable because they have to estimate the values of infinite actions and states. Naturally, since our problem requires us to simulate real-world states and actions, we have explored the implementation of various Policy Gradient algorithms. The 4 different Policy Gradient methods we have implemented are: REINFORCE, PPO, DDPG and SAC.

4.2.1 REINFORCE

4.2.1.1 Algorithm 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 manner.

During a given episode, the neural network would output a distribution $N(\mu, \sigma^2)$ for each action space item for a given state. A value would then be sampled from each of the 4 distributions, thus determining the action the agent would take. A reward would be collected for the action taken and stored along with the probability of the action taken. Once the episode/trajectory was completed, the loss for each step would be calculated.

The loss function for REINFORCE, a vanilla Policy Gradient implementation, 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 the case of REINFORCE is simply the cumulative discounted rewards.

4.2.1.2 Hyperparameters In our various training runs for REINFORCE, we used the Adam Optimizer with a learning rate $\eta = 0.001$. The discount factor γ was set to 0.9, since we wanted the agent to understand that actions had long-lasting affects or consequences. This would be especially important when the aircraft would make maneuvers that would be at once detrimental and hard to recover from, e.g. inverted dives.

4.2.1.3 Network Architecture During the course of our experimentation, we changed the number of nodes and layers but ultimately finalized a 4 layer architecture: 1 input layer, 2 hidden layers and 1 output layers. The observation space information is fed into the input layer so the number of input nodes equal the number of input parameters, which in our case was 8. Both hidden layers had 256 nodes. The output layer provided us with information of the probability distribution for each action space item, i.e. a μ and a σ value for 4 different action space items, totalling 8 output nodes. Hidden Layer 1 and 2 used the Rectifier Linear Unit (ReLU) activation functions, and the output layer employed two different activation functions. Since the value of the standard deviations can not be below 0, we used the sigmoid activation function for the σ values. The value of the mean can range between -1 and 1, i.e. the range of values for each action space item, we used the tanh activation function for the μ values.

4.2.2 Deep Deterministic Policy Gradient (DDPG)

4.2.2.1 Algorithm 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 [10]. 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 [9] 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:

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

4.2.2.2 Network Architecture 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.

4.2.2.3 Hyperparameters Policy Gradient methods are notoriously sensitive to hyperparameters. 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.

4.2.3 Proximal Policy Optimization

4.2.3.1 Algorithm Proximal Policy Optimization like DDPG also belongs to the class of 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.

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 far from the original policy, preventing 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 1 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:

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

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 $1+\epsilon$, ϵ being the hyperparameter, which in this paper is set to 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.

4.2.3.2 Network Architecture PPO was implemented for continuous a well as discrete action spaces. The primary architecture for actor networks for both the problems was same except for the output layer. The actor network that we utilized consists of 3 hidden layers. Input layer has 8 units to handle observation space, each hidden layer has 256 units. Output layer for continuous action space has 8 units representing 4 μ and 4 σ for all the actions. Whereas, for discrete action space we trained a separate agent for predicting each action. The output layer in this case consists of 11 units representing output values for an action. We use $ReLU$ activation function in hidden layers and in the output layer 4 units use $tanh$ for limiting μ values between -1 and 1, whereas the remaining 4 units use $ELU + 0.0001$ to keep standard deviation from becoming negative.

4.2.3.3 Hyperparameters Although PPO was implemented both in continuous as well as discrete action spaces, however, the hyperparamters were kept same throughout. A learning rate of $\alpha = 0.0003$ was used. The discout factor γ was set to 0.99 and $\lambda = 0.95$ for generalised advantage estimation. Policy clip of $\epsilon = 0.2$ was used in order to calculate the surrogate objectives. Most of the hyperparameters were kept same as those in PPO paper. Since our experiment had a very bad sample efficiency and it also took longer than 1 minute to run 1 episode. Therefore, we decided to use a $buffer$ of size 5000 and a $minibatchsize = 100$. Adam optimiser was used to backpropagate the loss values for 70 epoch during a single training iteration. Buffer size of 5000 was chosen keeping in mind that the agent should have at least more than 1 episode of data so that it is able to learn from experience of multiple episodes but at the same time the buffer size was small enough to let the agent do more frequent learning steps to increase its performance.

4.2.4 Soft Actor-Critic

Model-free deep reinforcement learning (RL) algorithms have been shown to be capable of solving a 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 demands a very detailed and intense hyperparameter tuning. Both of these challenges severely limit the applicability of such methods to complex, real-world domains. Soft actor-critic 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.
In Soft Actor-Critic algorithm we consider a parameterized state value function $V_\psi(s_t)$, soft Q-function $Q_\theta(s_t, a_t)$, and a tractable 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, and the policy as a Gaussian with mean and covariance given by neural networks. The gradient equations for the above parameters can be written as below.
for the value network:

$$\begin{aligned} \hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(s_t)(V_\psi(s_t) - Q_\theta(s_t, a_t) \\ + log\pi_\phi(a_t|s_t)) \end{aligned}$$

where the actions are sampled according to the current policy, instead of the replay buffer.

for the critic network:

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(s_t, a_t)(Q_\theta(s_t, a_t) - r(s_t, a_t) - \gamma V_{\overline{\psi}}(s_{t+1}))$$

The update makes use of a target value network $V_{\overline{\psi}}$, where $\overline{\psi}$ can be an exponentially moving average of the value network weights, which has been shown to stabilize training. However, in our implementation we use an update smoothing constant which mimics soft update. The actor policy can be updated using the following equations:

$$a_t = f_\phi(\epsilon_t; s_t)$$

using the above defined a_t:

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi log\pi_{phi}(a_t|s_t) + (\nabla_{a_t} log\pi_\phi(a_t|s_t) - \nabla_{a_t} Q(s_t, a_t))\nabla_\phi f_\phi(\epsilon_t; s_t)$$

where a_t is evaluated at $f_\phi(t; s_t)$.

4.2.4.1 Network Architecture Our implementation of SAC has 1 actor network, 2 critic networks, 1 value network and 1 target value network. Number of layers in all the networks is same; input layer followed by 2 hidden layers, of 256 units each, followed by an output layer. Number of units in the input and output layers of actor, critic and value networks differ. Actor network has 3 units in the input layer to handle a simplified observation space of 3 parameters and output layer consists of 8 units to predict μ and σ for 4 actions. Critic network has 7 units in the input layer to handle 3 parameters from observation space and 4 actions and 1 unit in the output layer to predict Q-value for the given state-action pair. Whereas, value network has 3 units in the input layer and 1 in the output layer to predict value function of the given state.

4.2.4.2 Hyperparameters The soft actor-critic algorithm makes use of reward scaling as well as soft target update to fine tune the update of target network. For all the networks we started with learning rate $(\alpha \,\&\, \beta) = 3 \times 10^4$, target value update smoothing constant $(\tau) = 0.005$, reparameterisation noise for $\sigma = 1 \times 10^{-6}$. Initially we used a reward scale of 1 which acts as a multiplier for the rewards that the agent collects during an episode, this gives more weight to the rewards as compared to the Q-value and value function while backpropagating the loss for the critic network. By tuning the reward scale we can balance the exploration-exploitation nature of the agent. The agent did not explore a lot using the reward scale of 1. Therefore, we changed it to 10 in the second training iteration. The buffer was set to 1×10^6 steps, with a batch size of 1024 and number of epochs to 5. After each step, the agent would perform gradient descent 5 times using a randomised batch from the experience replay buffer. This greatly improved the learning behaviour.

5 Results and Analysis

5.1 REINFORCE

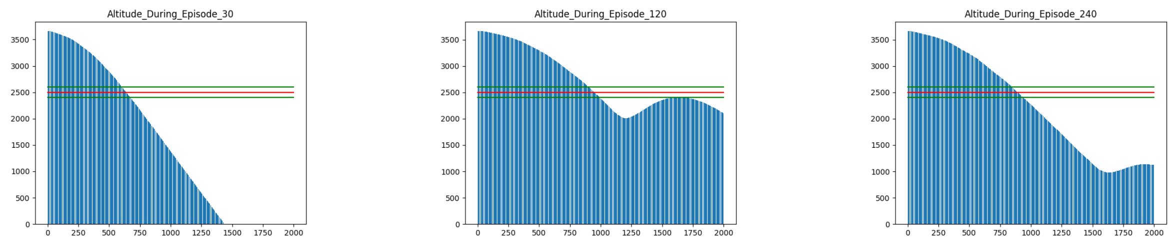
Discussed here is the most promising run of our REINFORCE algorithm. Initially, the agent crashes the plane consistently, as is observed in Figure 3a, however it eventually learns to not crash and then eventually also starts trying to come back to 2500m after it has descended past it, as observed in Figure 3b. However, instead of improving upon this, we see in Figure 3c 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.

5.2 DDPG

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.

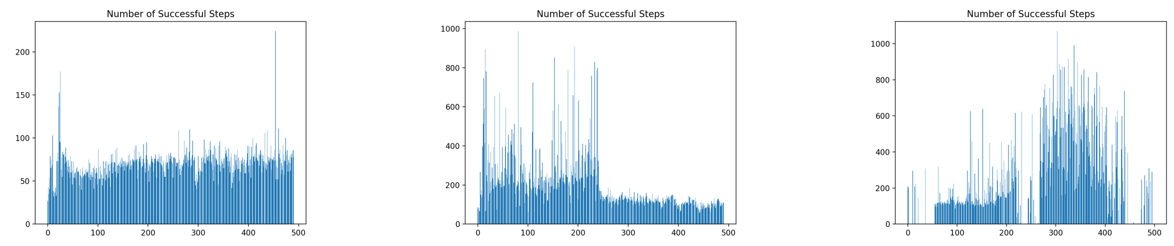
In our initial implementation, we observed a lot of promising results. In the first 500 episodes of training, the plane keeps crashing, as is shown in Figure 5a. However we see a marked improvement in the flight trajectory as we get closer to 2000 episodes of training, confirmed by Figure 5c.

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



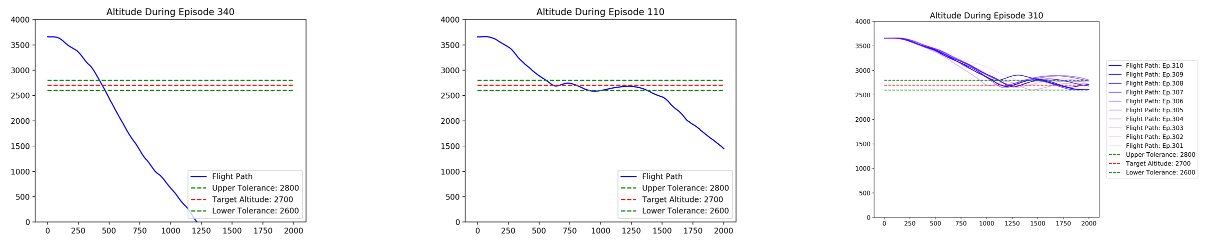
(a) Episode 30 (b) Episode 120 (c) Episode 240

Figure 3: REINFORCE: Altitude During Episode



(a) Episode 0 - 499 (b) Episode 500 - 999 (c) Episode 1500 - 1999

Figure 4: DDPG - Number of Successful Steps



(a) Episode 0 - 499 (b) Episode 500 - 999 (c) Episode 1500 - 1999

Figure 5: DDPG - Altitude During Episode

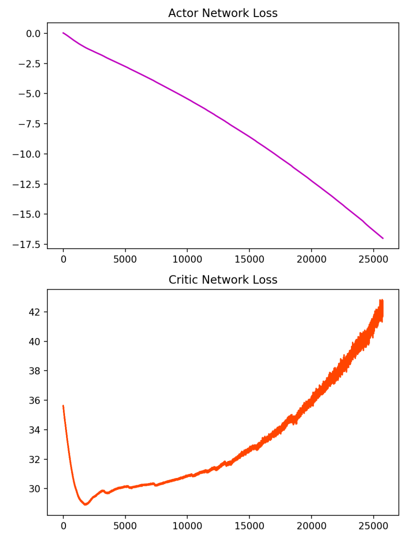
structure.

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

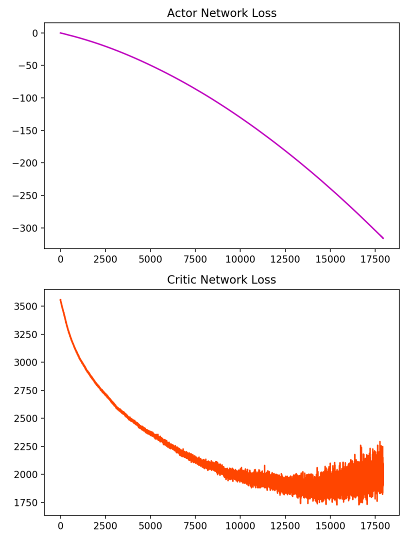
Our focus now was to use the loss functions of the Actor and Critic networks as guides to improve the performance. By tweaking the learning rate we were able to see an improved performance in terms of loss, however the agent's performance still did not achieve satisfactory levels. We tried different 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 6a to Figure 6b. 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.

5.3 PPO

PPO was implemented in different setting w.r.t. the number of networks and type of action space. Hyperparameters for all the versions of PPO are given in the index.



(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 6: DDPG - Actor, Critic Network Losses

5.3.1 Continuous Action Space with 1 Network

Initial implementation was for continuous action space and the network returns parameters for 4 normal distributions corresponding to the 4 actions: latitudinal stick, longitudinal stick, rudder pedals and throttle. Actions were sampled from the normal distributions to introduce randomness. However, the results were inconclusive. Average scores for the last 100 episodes did not show a lot of variation, as can be seen in Figure 8b and the descent trajectory that the aircraft followed did not change either over the course of 500 episodes, as shown in Figure 7. We realized that the underlying state space was infinitely large for the agent to learn it in a limited time.

5.3.2 Discrete Action Space with 4 Networks

Following the results of PPO in continuous action space, the action space was discretized into 11 equal segments with a step size of 0.2 for the first 3 actions and with a step size of 0.1 for throttle. Furthermore, a separate network was used for each action and all the networks were trained separately. The output layer of each network consisted of 11 units and a softmax activation function which gives a Categorical probability distribution over the discretized action.

Despite early crashes, the agent learnt to avoid them quickly. Just after 500 episodes, it was able to stay inside the target zone for more than 600 steps on average, as shown in Firgure 9a. The flight trajectory figure below shows the altitude during episode 1850 (although the plot title says episode 40, but that is because plot data was not being saved during each training run). The agent was trained for over 2000 episode with satisfactory results for episodes of 2000 steps. However, when the agent was tested, it could not maintain altitude after 2000 steps. We realized that although the agent was showing a positive trend towards the behavior it was supposed to learn, however, it still had not fully learned. An episode with 2000 steps was simply not long enough for it experience a wide array of states.

5.4 Soft Actor-Critic

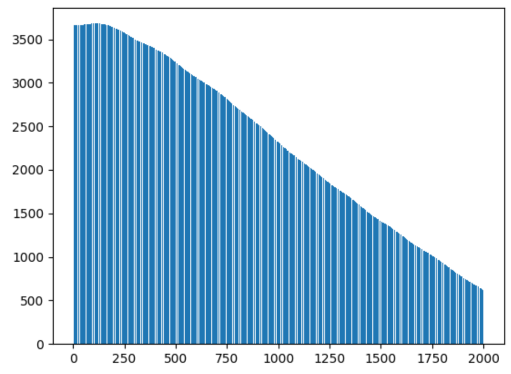
We performed 2 iterations of Soft Actor-Critic algorithm by changing some hyperparamters. The second iteration showed good results.

5.4.1 Reward Scale = 1

During the training the agent seemed very conservative in exploration despite being formulated on entropy maximization framework. In Figure 10 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.

5.4.2 Reward Scale = 10

For the second training iteration we had very limited amount of time available, therefore, to make


(a) Episode 10

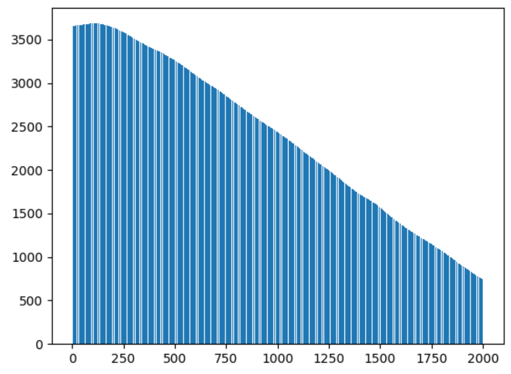
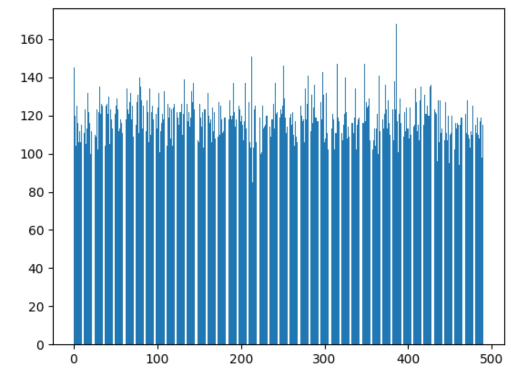
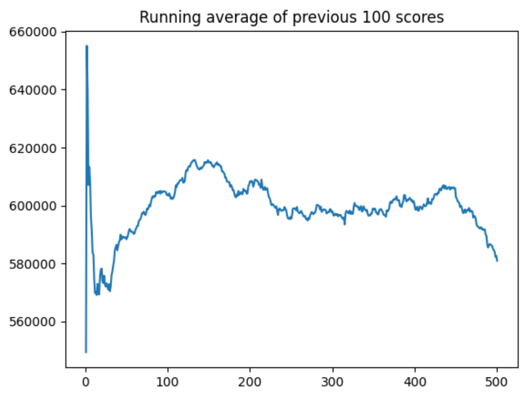

(b) Episode 490

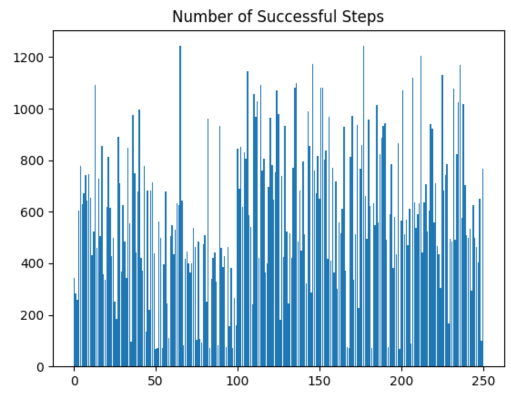
Figure 7: PPO - Altitude During Episode


(a) PPO - Number of Successful Steps

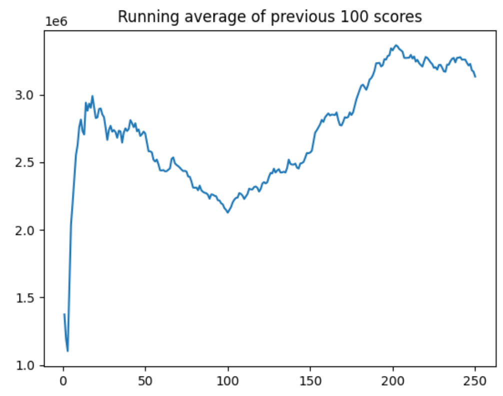


(b) PPO - Avg score of last 100 episodes

Figure 8: PPO - Continuous Action Space



(a) PPO - Number of Successful Steps

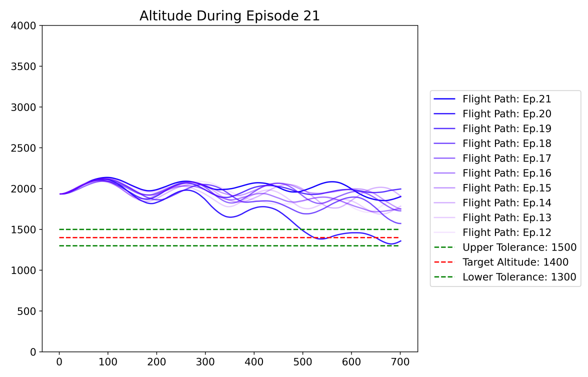


(b) PPO - Avg score of last 100 episodes

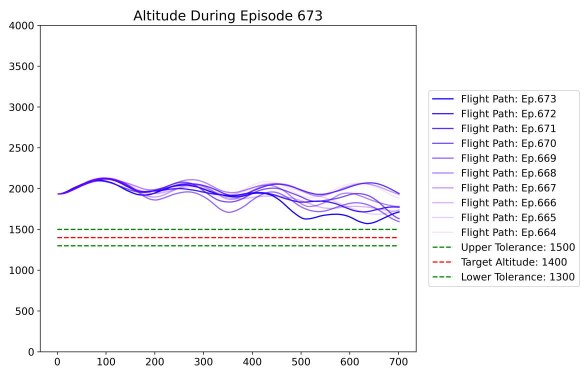
Figure 9: PPO - Discrete Action Space

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 $\pm 10^\circ$.

We can observe from above figures that the agent not only started exploring more, but it also

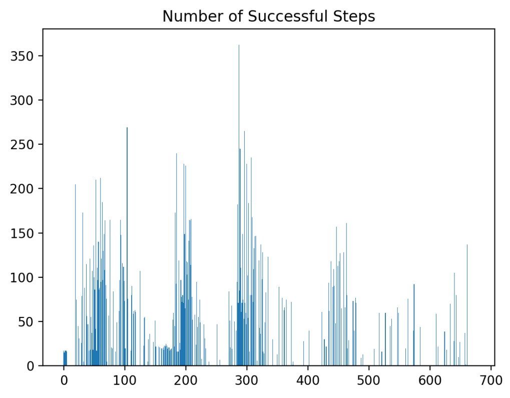


(a) Episode 11 - 21

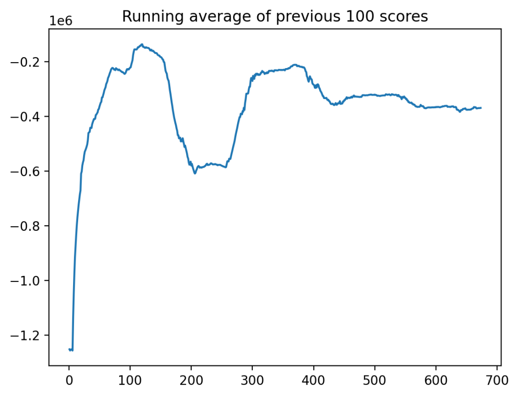


(b) Episode 663 - 673

Figure 10: SAC - Altitude During Episode

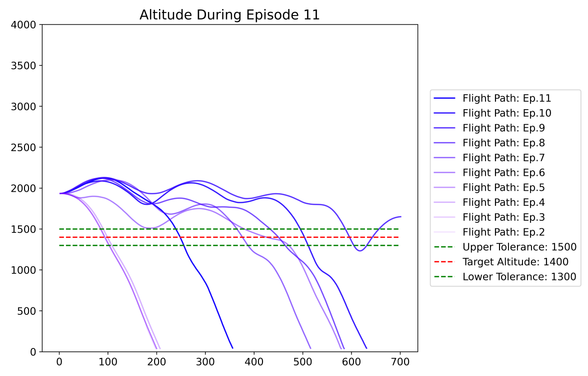


(a) SAC - Number of Successful Steps

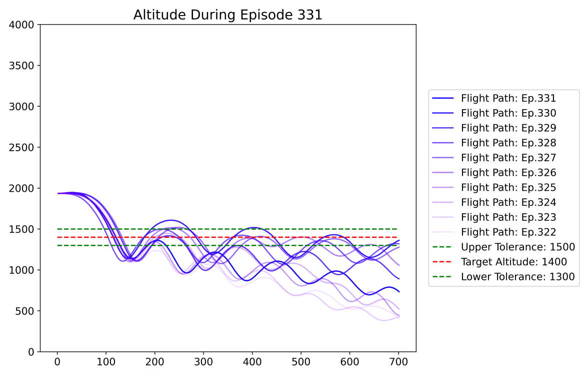


(b) SAC - Avg score of last 100 episodes

Figure 11



(a) Episode 1 - 11



(b) Episode 321-331

Figure 12: SAC - Altitude During Episode

recovered from crashed very quickly, since the reward was being weighted more while performing gradient descent of critic network. The model was trained followed by 3 tests where the initial altitude was 1836 m.

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

The agent **successfully** accomplished the above tests. Results of the above tests were shown in the form of a demo video.

6 Discussion

6.1 Limitations

The earlier phase of the project presented us with a very steep learning curve as we had a very limited experience of working with RL problems. The second challenge that we faced was that there was no open source environment wrapper available for X-Plane 11. Therefore, significant time and efforts were spent on writing and debugging the environment ourselves using *X-Plane Connect*.

A reinforcement learning agent can take days to train even for seemingly easier tasks in simpler domains but when the underlying agents are complicated and close to reality like X-Plane 11, training time requirement can increase exponentially. We were limited by compute resources as well as by the number of training instances that we could run in parallel since a separate license needs to be purchased to run each instance of X-Plane 11. Therefore, we could not benefit from asynchronous versions of actor-critic algorithms.

6.2 Future Work

In this work different approaches are presented 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.

7 Conclusion

Our objective for this project was to experiment with different Reinforcement Learning algorithms, to determine the best performing approach in order for an Intelligent Autopilot System to be able to learn various flight maneuvers. We broke our goal further down to first learning to change altitude as desired. With this target determined, we implemented 4 different RL algorithms: REINFORCE, PPO, DDPG and SAC. Our discussion in **Section 5** analyzes the performance of these algorithms. REINFORCE, it was determined, was not adequate for our application since it was computationally inefficient and the scope of the problem was too large. PPO and DDPG showed promising results with some training cycles displaying desired behaviour. However, ultimately the training of both of these algorithms did not converge to a perfect model. In the end, it was SAC that was able to achieve the goal of changing aircraft altitude to maintain it around the desired target altitude, including both descent and ascent scenarios.

Through this project, we were able to fully immerse ourselves within the realm of Reinforcement Learning and Policy Gradient methods, and have been increasingly motivated by the results we encountered, including successfully achieving our target objective. Moving forward we will continue to work on the tasks delineated in **Section 6.2**, with the goal of achieving an improved and realistic autopilot agent and contributing positively to the space of IAS and RL research.

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] Mario S Holubar and Marco A Wiering. "Continuous-action Reinforcement Learning for Playing Racing Games: Comparing SPG to PPO". In: *arXiv preprint arXiv:2001.05270* (2020).

[8] 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).

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

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

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

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

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

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