USP Independent Study Module (ST)

Group Report

Deep Learning and Reinforcement Learning: Embedding Artificial Intelligence into a Flapping-Wing MAV

Student Names, Module Code:
Arijit Dasgupta, UIS3902S
Jasper Tan, UIS3903S
Chong Yu Quan, UIS3902S

Student Number:
A0182766R
A0136232R
A0136286Y

Student Emails:
arijit.dasgupta@u.nus.edu
jasper.tan@u.nus.edu
chong.yuquan@u.nus.edu

April 16, 2021

Contents

List of Figures

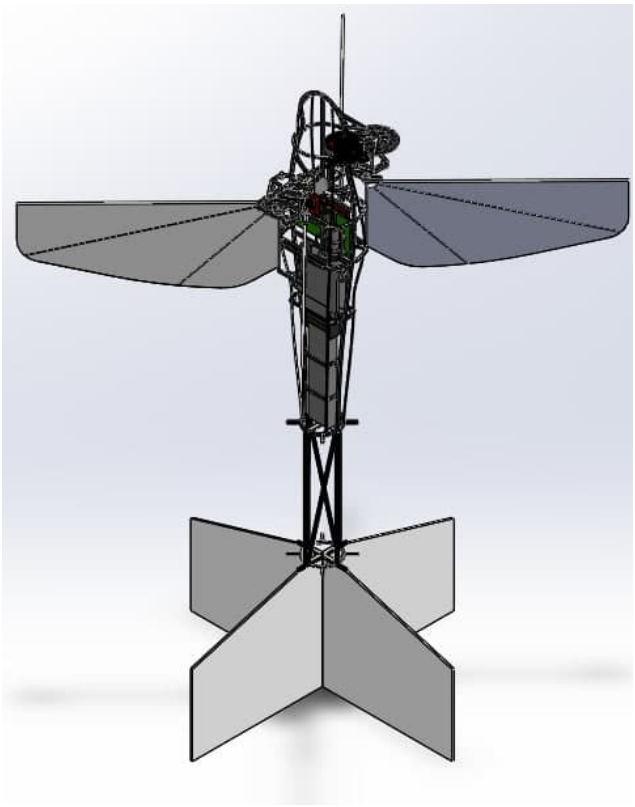
1 Introduction

It is a bright summer day in the year 1956. Eleven scientists and mathematicians gather at a college in Hanover, New Hampshire. The have the entire floor of the Dartmouth Mathematics department to themselves. They start discussing about symbolic representations and debating over inductive and deductive systems. It may seem like a rather innocuous meeting to them, but what they do not know is, this very event is to be known to mankind as the birth of Artificial Intelligence (AI). The likes of Claude Shannon, Marvin Minsky & Herbert Simon were in that very room. It was the first time in history when a group of international & prominent experts in diverse fields came together to discuss the study of AI. Their vision and initiative have come a long way to mould into the AI we see today. With the new invention of computer systems that could perform calculations multiple times faster than humans, it seemed like the next natural step. Promises of robotic assistants, artificial reasoning, flawless natural language processing and conscious minds were made and many in the 60s and 70s were optimistic of these goals. However, people waited and waited and little was delivered. The AI winter (a period with a lack of funding and optimism in AI research) eventually came in the 1970s and stuck around for a long time. Many back then attribute this to inferior computational capabilities and the lack of any visible progress in AI.

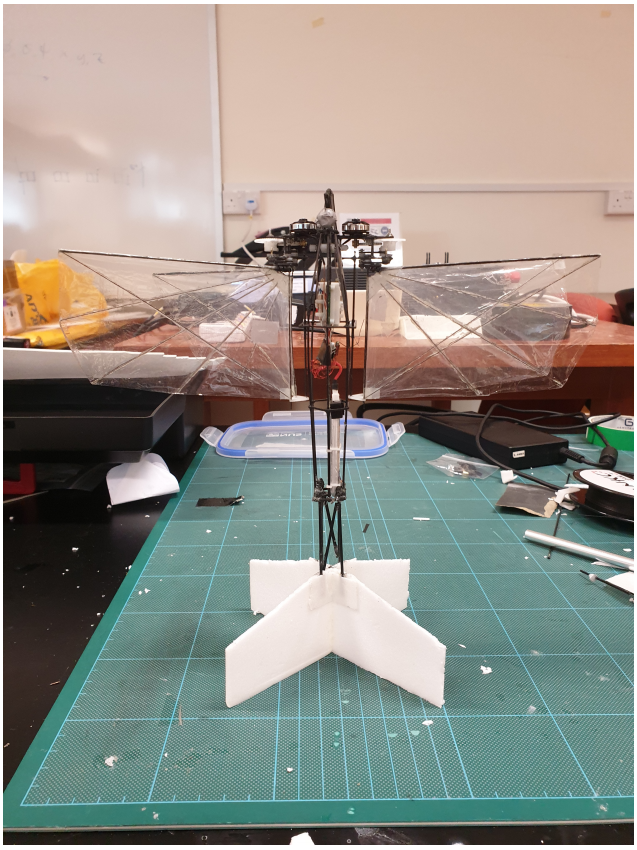
However, this changed very quickly in the 2010s when the computational power in processors finally caught up to the deep learning methods developed by statisticians and computer scientists. The deep learning revolution paved the way for the new AI wave we see today. Companies and Institutions use object detection, natural language processing and prediction models all the time to save costs and make better decisions. Millions and Billions of dollars are spent to optimise systems with deep learning and AI. Switching to the field of mechanical engineering, there have also been numerous developments in robotic and autonomous systems. The bridge between mechanical engineering and artificial intelligence is closing up everyday. The upheaval of AI and deep learning into hardware and mechanical systems has become the bleeding edge of technological innovation.

As students of this age with a keen interest in mechanical engineering and computer science, we aim to embrace the best of both worlds in this project. Our objective is to design a real-time system and training setup to train an artificial agent to control & fly a Flapping Wing Micro-Aerial Vehicle (MAV). This report will highlight the prerequisites: theory of reinforcement learning, hardware training setup, casting the problem into a Markov Decision Process & methods of training. A computationally intelligent agent will be given the power to actuate the controls of the MAV and learn to fly it with a deep reinforcement learning algorithm. Figure 1.1a & 1.1b show the CAD model and actual physical model of the MAV. The wings of the MAV can be controlled using two independent motors and roll, pitch & yaw control can be achieved by tilting the wings. The tilt can be controlled by two linear

servos. Refer to our ME3103 design report for the full details of the mechanical design and testing of the MAV. In this project, we shall assume that we have a fully functional MAV that can be controlled.



(A) CAD Model



(B) Actual Physical Model

FIGURE 1.1: MAV referred to in the present project

2 Deep Reinforcement Learning: Theory and Algorithm

2.1 Reinforcement Learning

To model the behaviour of the AI, we decided to take a reinforcement learning approach. In the reinforcement learning framework, we define an agent as an individual entity that can take actions based on an observed state. We define the state as a description of the thing an agent is controlling (e.g. position, speed, orientation of a robot). In most reinforcement learning problems, the agent is usually the same as the thing it is controlling (e.g. grid-world, travelling salesman etc.) hence the term "the agent's state" is commonly used. However, it is not always the case (board games, robotics etc.), like in this project (explained in section 4.1).

Actions refer to decisions made my the agent to potentially alter the state of what it is controlling. Given any action taken at a state, the agent always receives an immediate reward. The reward can be defined in any way where negative means bad, positive means good and 0 means nothing. The reward structure can be custom defined to suit the purpose of a goal. For instance, if we want the MAV to fly up, an upward velocity can be given a positive reward. The agent resides in an environment. The environment is what changes the state of the thing the agent controls based on a given action. We can define all behaviour of an agent by a sequence of state, action & rewards as shown by the relation in Figure 2.1.

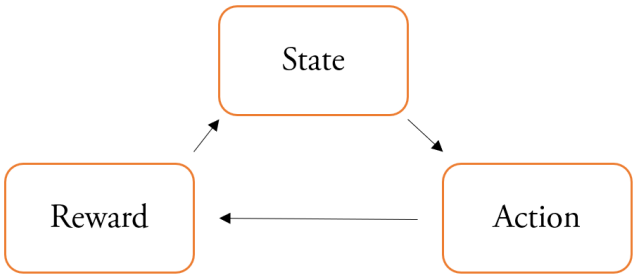


FIGURE 2.1: Sequential relationship of state, action & reward

The next step is to make the Markovian assumption of our agent. The Markovian assumption states that the next state, S' is only dependent on the current state, S and the action taken in that state, A. This is means that none of the future states and actions are conditional in the past. The current state is sufficient to describe the situation of the agent and none of the previous states matter. This memory-less property is essential to map our MAV agent into a Markov Decision Process (MDP). A

MDP is a map of states, actions and state-transitions for a given agent & environment. In the case of the MAV, we will justify the Markov assumption in section 4.2 after defining the states. An example of a MDP is illustrated as shown in Figure 2.2.

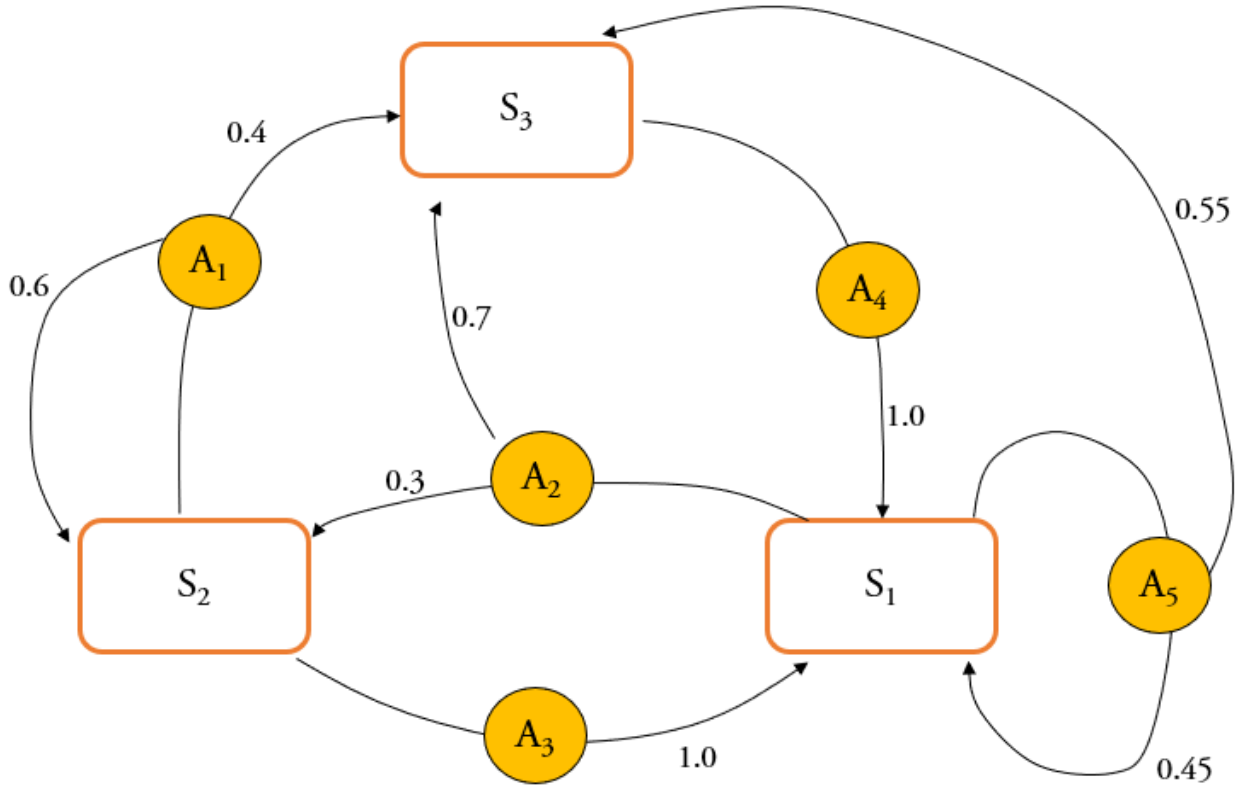


FIGURE 2.2: Example of a MDP

The MDP in Figure 2.2 shows three possible states the Agent can be in (S_1, S_2 & S_3). In different states, the agent can take different actions. Each action has different probabilities of reaching different states. For instance an agent in S_3 that takes action A_4 will reach the state S_1 100% of the time, but an agent at S_2 taking action A_1 only has a 0.4 probability of reaching S_3 and 60% of the time it will return to its own state. We refer to these probabilities as the state transition probability, $f(S', A, S)$ which is the probability that taking action A at state S will lead to a transition to S'. Like all probability rules, the sum of probabilities for all existing S' for a given A & S add up to 1, $\sum^{S'} f(S', A, S) = 1$. Note that S' can be the same as S, like the transition $S_1 \rightarrow A_5 \rightarrow S_1$. I is important to define these probabilities to represent the stochastic nature of the environment. Taking the same action in the same state may not always lead to the same subsequent state, mimicking real-world conditions.

We now define the policy. The policy can be thought of as the decision maker of the agent, which is essentially the probability of choosing action A given S. This is commonly represented as $\pi(A \mid S)$. We are able to define these functions with few variables due to the Markov assumption of MDPs. The next step is to define the expected reward function with different domains. Equation 2.1 define the state-dependent expected reward function. Note that R refers to the random variable for reward while r refers to the scalar reward for this specific equation.

$$\mathbb{E}[R \mid S] = \sum_r r\rho(r \mid s) \tag{2.1}$$

We can also define a state-and-action dependent reward function as shown in Equation 2.2.

$$\mathbb{E}[R \mid S, A] = \sum_r r\rho(r \mid s, a) \tag{2.2}$$

Finally, we define the four-argument probability function that tells us the probability of getting reward R in a $S \to A \to S'$ transition as shown in Equation 2.3.

$$\rho(R, S' \mid S, A) = f(S', A, S) \cdot \rho(R \mid S') \tag{2.3}$$

The next step is to define the notion of return. The return, G_t, can be thought of the total reward that the agent can get by taking actions and transitioning states in the long run. A discount factor, $\gamma \in [0, 1]$, is used to indicate to the agent that immediate rewards matter more than rewards in the future. This is represented in Equation 2.4 where t refers to the current time-step.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

We can thus define G_t recursively as in Equation 2.5. The equation can be intuitively understood by realising that the long-term total reward in the current time-step is equal to the reward in the next time step plus the discounted total long-term total reward in the next time step.

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.5}$$

Under a given policy, π, we can define the expected long-term rewards, G_t, based on either the "agent's state" only or the "agent's state" and action. They are called $V_\pi(S)$ and $Q_\pi(S, A)$ respectively. Both terms are intuitively understood as the "goodness" of being in a state, or taking an action in a given state. The latter is commonly known as the Q-value of taking an action in a given state. Both terms are mathematically defined in Equations 2.6 and 2.7.

$$V_\pi(S) = \mathbb{E}_\pi[G_t \mid S] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S] \tag{2.6}$$

$$Q_\pi(S, A) = \mathbb{E}_\pi[G_t \mid S, A] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S, A] \tag{2.7}$$

After some mathematical manipulation based on all the functions and relations defined so far, we arrive at the famous Bellman equation. Equation 2.8 shows the Bellman equation for Q_π. Note that A' refers to the action taken in the subsequent time-step.

$$Q_\pi(S,A) = \sum_{S',R} \rho((R,S' \mid S,A)[R + \gamma \sum_{A'} \pi(A' \mid S')Q_\pi(S',A')] \tag{2.8}$$

The important part of the Q_π Bellman equation is the recursive relation between $Q_\pi(S,A)$ and $Q_\pi(S',A')$ which will come in handy for the Q-learning algorithm (described later on).

Now that we have defined the basic mathematics of reinforcement learning, we now define the goal. The aim of reinforcement learning is to find an optimal policy, $\pi^*(A \mid S)$, that gives the maximum possible long-term discounted reward. For the purposes of this project, we focus only on $Q_\pi(S,A)$ instead of $V_\pi(S)$. Hence we aim to find $Q_{\pi^*}(S,A)$. The optimal policy can be defined in a greedy manner, which is to say that for all actions A for a given state S, the A that provides highest $Q_\pi(S,A)$ value is the optimal action to take in any state.

There are numerous approaches to find an optimal policy in reinforcement learning. One philosophy is using model-based techniques like dynamic programming. However, such an approach requires the entire MDP to be known. For many real-time physical systems like the MAV, finding the model is incredibly difficult and can be deemed impossible. The best approach would be to use temporal difference. The most popular and effective algorithm is known as Q-learning. This algorithm tracks a series of state, action, rewards & aims to learn the $Q_\pi(S,A)$ for all state-action pairs. A Q-table is created with every row being a state and every column being an action. The Q-learning algorithm is able to update its $Q_\pi(S,A)$ values after every time step, where it uses a rollout of S,A,R,S',A'. As seen in the Bellman equation, there is a mathematical link between the $Q_\pi(S,A)$ values of state and actions in consecutive time steps. The update rule for Q learning is derived from the Bellman equation and based on a learning rate update of Q values. A learning rate update means that the updated value of a quantity is equal to the sum of a portion, α, of the new value and (1 - α) of the old value. The update rule is as shown in Equation 2.9.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_{A'} Q(S_{t+1},A') - Q(S_t,A_t)] \tag{2.9}$$

During training, at every time-step, the agent usually chooses a greedy action (as defined earlier) based on its state and carries it out. However this bring about the dilemma of exploration and exploitation in reinforcement learning. Taking the greedy action is like an agent exploiting its sense of the optimal action. However, if one only takes greedy actions, they may miss out on many other possible actions that are potentially more optimal. Hence there is a need for exploration, where the agent will choose a random action and execute it. The agent cannot explore too much as learning will slow down. Hence there must always be a fine balance between exploration and exploitation. This concept is universal across all domains and algorithms of reinforcement learning and is a fundamental aspect of any training process. We can define ϵ, which is the probability for an agent to take a random action in any time-step. The general technique that one usually employs is to start by exploring heavily at the start ($\epsilon = 1$), as the agent is yet to see most existing states. Then as the training improves and the agent learns, it can start to exploit more and the ϵ reduces to virtually 0. In the case of this project, the algorithm used employs this idea of exploration in a different way.

Understanding the intuition behind the mathematics of basic reinforcement learing and Q-learning is only part of the theory that we need to start training the MAV with AI. One big downside of Q-learning is that when an agent has thousands or millions of possible states and actions, the Q table becomes impractically large. This especially the case as Q-learning an other traditional reinforcement learning algorithms make the assumption that all actions and states are discrete (As seen in the MDP in Figure 2.2). However, in special cases like this project, both state and actions (Chapter 4) are in the continuous space. Hence, there is a need to replace the Q table with a value function approximator. This is how we bring deep learning into the picture.

2.2 Deep Learning

As stated in the previous section, for most realistic reinforcement learning problems, a value function approximator is required to estimate the value function. The value function approximator must be able to generalise across seen states / state-action pairs to unseen states / state-action pairs. As neural networks are found to be universal function approximators from work of Hornik et al. [1], they are selected to be the value function approximators for most deep reinforcement learning problems and are the staple architecture utilised for most state of the art deep reinforcement learning algorithms. For example, by incorporating neural networks to Q-learning algorithm as stated in the previous section with the concept of a experience replay buffer to be further elaborated, the resultant algorithm is obtained is the Deep Q-Learning (DQN) algorithm by Mnih et al. [2] which was one of most notable and successful early deep reinforcement learning model. Models trained using the DQN on seven Atari 2600 games from the Arcade Learning Environment was found to outperform all previous approaches on six of the games and surpasses a human expert on three of them. By referring the neural network value function approximator with weights θ as a Q-network, the Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i as shown in equation 2.10, where ρ_π is the state visitation distribution for a policy π and E is the environment.

$$\begin{aligned} L_i(\theta_i) &= \mathbb{E}_{S\sim\rho_\pi, A\sim\pi, R\sim E}[(y_i - Q(S, A|\theta_i))^2] \\ y_i &= \mathbb{E}_{S'\sim E}[r(S, A) + \max_{A'} Q(S', A'|\theta_{i-1})|S, A] \end{aligned} \tag{2.10}$$

It is important to note that the weights from the previous iteration, θ_{i-1}, are held constant when optimising the loss function,$L_i(\theta_i)$, and that the targets are dependent on θ unlike supervised learning. Given that neural networks are differentiable function approximators, by differentiating $L_i(\theta_i)$ with respect to θ_i, the following gradient as shown in equation 2.11 is obtained.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{S\sim\rho_\pi, A\sim\pi, R, S'\sim E}[r(S, A) + \max_{A'} Q(S', A'|\theta_{i-1}) - Q(S, A|\theta_i))\nabla_{\theta_i} Q(S, A|\theta_i)] \tag{2.11}$$

It can be observed that the algorithm learns off-policy givens that it follows a behavioural policy, π, to ensure adequate exploration, while learning about the greedy policy given that the argmax was used in the determining the gradient as shown in equation 2.12. The behavioural policy is typically

an ϵ-greedy policy where the greedy strategy is followed probability $1-\epsilon$ and selects a random action with probability ϵ. With the gradient, $\nabla_{\theta_i} L_i(\theta_i)$, gradient descent (e.g. stochastic gradient descent), can be utilised to update the Q-network parameters, θ, through back-propagation to minimise $L_i(\theta_i)$ as shown in equation 2.12, where α is a step-size hyperparameter known as the learning rate.

$$\begin{aligned} \Delta\theta_i &= -\alpha \nabla_{\theta_i} L_i(\theta_i) \\ \theta_{i+1} &\leftarrow \theta_i + \Delta\theta_i \end{aligned} \tag{2.12}$$

In the scenario where weights, θ, are updated after every time-step and the expectations are replaced by single samples, it can be observed that the resultant algorithm is the Q-learning algorithm with neural networks. However, given that stochastic gradient descent using single samples is extremely sample inefficient despite its simplicity, DQN utilises a technique known as experience replay where the agent's experiences at each time-step, $E_t = (S_t, A_t, R_t, S_{t+1})$, are stored in a replay buffer, $D = E_1, E_2 \dots E_N$. Hence in prose, the DQN algorithm generally involves taking action, A_t according to ϵ-greedy policy, storing experience $E_t = (S_t, A_t, R_t, S_{t+1})$ in replay buffer, D, and sampling random mini-batches of experiences from D to compute Q-learning targets using θ_{i-1}. Thereafter, the updated loss function is optimised as shown in equation 2.13 before performing a variant of stochastic gradient descent (e.g. RMSPROP, Adam).

$$L_i(\theta_i) = \mathbb{E}_{S,A,R,S' \sim D}[(y_i - Q(S, A|\theta_i))^2] \tag{2.13}$$

2.3 Deep Deterministic Policy Gradients (DDPG)

2.3.1 Policy Gradient

The algorithms stated in previous sections have been value based, where the policy is generated directly from the Q values generated from the value function approximator that is the Q-network (e.g. greedy, ϵ-greedy). While the merits of value based reinforcement learning have been exemplified by the example of the DQN in the context of the Atari games, it is important to note that Atari environment is considerably much simpler that reality given that the action space of the game is discrete, constraint by the inputs of the joystick for the Atari. However, in the context of the MAV, the action space is continuous given that the inputs for the motors and linear actuators are real numbers. Given that in value based learning, a Q value must be attributed to a specific state-action pair, it is practically unfeasible to represent the entire action space through Q values as that would require an infinite number of Q values as outputs of the Q-network. To address this problem, a possibility would be to manually discretise the action space such that the number of actions are reasonably finite. Naturally, the apparent drawback of such a solution would be the fact that the methodology of discretisation (e.g. number of discrete states, the discrete actions itself) may inherently remove the optimal policy from the action space. In the worst case, the agent may fail to perform at all instead of performing sub-optimally. In the context of the MAV, it is possible that by discretising the action space manually, it would have been impossible for the MAV to take flight no matter how the model is trained.

Given the stated constraints of value based reinforcement learning, another class of reinforcement learning algorithms that are called the policy based learning algorithms are considered. Its advantages over value based reinforcement learning are the fact it has better theoretical and empirical convergence properties as well as the fact that it can handle high dimensional and even continuous action spaces. Furthermore, it can learn stochastic policies, which is the generalisation of deterministic policies and some problems cannot be solved by deterministic policies. However, policy based methods often converges to local optimum as opposed to a global one and evaluating a policy can be very inefficient and high variance.

Policy based methods bypasses value learning by directly learning the policy itself and can be seen as an optimisation problem to find the parameter θ that minimises the cost function $J(\theta)$. While there are approaches that does not utilise gradients (e.g. hill climbing, simplex, genetic algorithms), greater efficiency is often possible with gradient based methods, hence focusing on policy gradient reinforcement learning methods. Policy gradients algorithms iteratively search for a local minimum by descending the local gradient of policy with respect to θ as shown in equation 2.14, where $\nabla_\theta J(\theta)$ is the policy gradient and α is learning rate.

$$\begin{aligned} \Delta\theta &= -\alpha \nabla_\theta J(\theta) \\ \theta &\leftarrow \theta + \Delta\theta \end{aligned} \tag{2.14}$$

By assuming that the policy, $\pi_\theta(S, A)$ is differentiable when $\pi_\theta(S, A) > 0$ and gradient $\nabla_\theta \pi_\theta(S, A)$ is known, the policy gradient can be computed analytically by using the following identity shown in equation 2.15. where $\nabla_\theta log \pi_\theta(S, A)$ is the score function.

$$\nabla_\theta \pi_\theta(S, A) = \pi_\theta(S, A) \frac{\nabla_\theta \pi_\theta(S, A)}{\pi_\theta(S, A)} = \pi_\theta(S, A) \nabla_\theta log \pi_\theta(S, A) \tag{2.15}$$

The policy gradient theorem generalises to all MDPs, stating that for any differential policy $\pi_\theta(S, A)$ and cost function $J(\theta)$, the policy gradient is as shown in equation 2.16.

$$-\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) * Q_{\pi_\theta}(s, a)] \tag{2.16}$$

In the REINFORCE algorithm that uses Monte Carlo gradient descent, the return $V_\pi(S)$ from a terminated episode is used as the unbiased sample of $Q_{\pi_\theta}(s, a)$. However, it was found that despite using unbiased value estimate, high variance is still exhibited in the policy gradient, which slows down learning. In addition, the learning process is extremely computationally expensive, requiring millions of episodes for convergence for simple tasks. Given the issues faced by Monte Carlo gradient descent, Actor-Critic policy gradient algorithms are introduced. Actor-Critic policy gradient algorithms maintains two sets of learned parameters as follows: 1) Critic: Updates ω parameters to estimate a value function (e.g. $Q_{\pi_\theta}(s, a) \approx Q_\omega(s, a)$), 2) Actor: Updates θ parameters to estimate policy $\pi_\theta(s, a)$. Actor-Critic policy gradient approaches follow an approximate policy gradient as shown in equation 2.17 below.

$$
\begin{aligned}
-\nabla_\theta J(\theta) &\approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_\omega(s,a)] \\
\Delta\theta &= \alpha * \nabla_\theta \log \pi_\theta(s,a) Q_\omega(s,a))
\end{aligned}
\tag{2.17}
$$

2.3.2 DDPG Algorithm

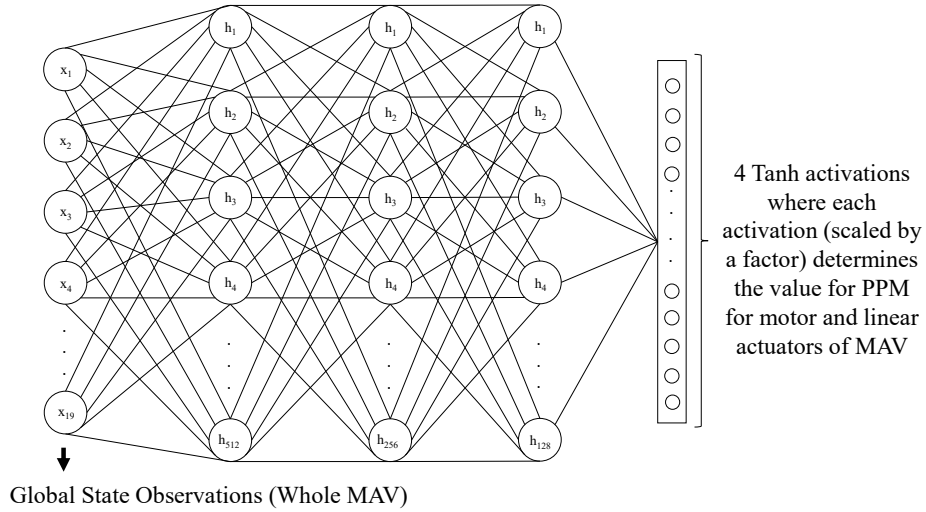


FIGURE 2.3: DDPG actor architecture

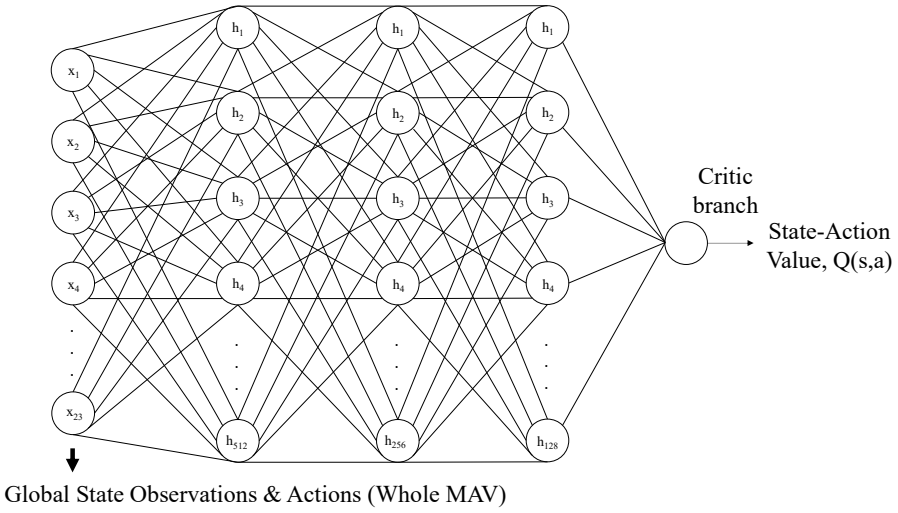


FIGURE 2.4: DDPG critic architecture

With the introduction of Actor-Critic policy gradient algorithms in the previous section, the DDPG algorithm can now be elaborated in detail. DDPG is an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces developed by Lillicrap et al. [3]. It consists of four fully connected deep neural networks (FCDNN), namely the actor FCDNN, critic FCDNN, target actor FCDNN and the target critic FCDNN, and a replay buffer. With inspiration from the DQN algorithm, the replay buffer have the same functionality to store the agent's experiences at each time-step, E_t, and the terminal flag as well. The replay buffer has a finite space (e.g. 1000000 episodes) and is based on a first in first out principle in replacing experiences stored in replay buffer. During the training process of the FCDNNs, a specified batch of the experiences stored in the replay buffer will be randomly sampled.

The actor FCDNN takes in global state observations as its inputs, where the input forward propagates through the FCDNN with three hidden layers of hidden units 512, 256, 128 respectively as shown in Figure 2.3. The number of outputs of the actor FCDNN corresponds to dimension of the action space of the reinforcement learning problem and have the Tanh as the activation functions. Given that the Tanh activation function is bounded between ± 1, the outputs of the actor FCDNN can be scaled by a specific constant to alter the range of the action variables in the continuous action space. The critic FCDNN takes in global state observations concatenated with the corresponding actions taken by the agent sampled from the replay buffer. The input forward propagates through the FCDNN with three hidden layers of hidden units 512, 256, 128 respectively to give a single output of the state-action value, $Q_\omega(s, a)$ as shown in Figure 2.4 above. The target actor FCDNN and target critic FCDNN follows an identical architecture with the actor FCDNN and critic FCDNN respectively. The two target FCDNNs are not utilised during experience gathering but for training of the actor and critic FCDNNs. The weights of each hidden unit in the target FCDNNs with are updated with based on a parameter, τ, with respect to its corresponding FCDNN counterpart as shown in equation 2.18 below.

$$\begin{aligned} \theta'_{target\ actor} &= \tau\theta_{actor} + (1-\tau)\theta_{target\ actor} \\ \omega'_{target\ critic} &= \tau\omega_{critic} + (1-\tau)\omega_{target\ critic} \end{aligned} \tag{2.18}$$

On initialisation of the FCDNNs, the target FCDNNs are a hardcopy of their original counterparts, i.e $\tau = 1$. After every subsequent training of the actor and critic FCDNNs, the target FCDNNs undergoes a softcopy update (e.g. $\tau = 0.005$). The training losses of the actor and critic FCDNNs are stated in equation 2.19 below.

$$\begin{aligned} -\nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_i \nabla_A Q_{\omega_{critic}}(S, A_{actor})\nabla_{\omega_{actor}}\mu_{\omega_{actor}}(S) \qquad (actor) \\ \delta_{\omega_{critic}} &= r + \gamma Q_{\omega_{target\ critic}}(S', A'_{target\ actor}) - Q_{\omega_{critic}}(S, A) \qquad (critic) \end{aligned} \tag{2.19}$$

For the critic losses with experiences from the replay buffer, the action of the target actor FCDNN from S', $A'_{target\ actor}$, is first obtained. From S' and $A'_{target\ actor}$, $Q_{\omega_{target\ critic}}(S', A'_{target\ actor})$ can be obtained from the target critic FCDNN, from which added to reward, r, gives the TD target. The critic loss is hence the mean squared error, δ^2_ω, between the TD target and the state-action value, $Q_{\omega_{critic}}(S, A)$, from the critic FCDNN. For the actor losses, the action selected with an updated policy

from the actor FCDNN, A_{actor}, is obtained. The actor losses are then the gradient of the state-action value, $Q_{\omega_{critic}}(s, A_{actor})$, with respect to A_{actor} using S and A_{actor} from the critic FCDNN multiplied with the gradient of actor FCDNN's policy output given S input, $\mu_{\omega_{actor}}(S)$, with respect to the actor FCDNN's parameters, ω_{actor}. Figure 2.5 highlights the pseudocode extracted from the work of Lillicrap et al. [3].

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

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

  end for
end for

FIGURE 2.5: Pseudocode for DDPG

3 MAV Training setup

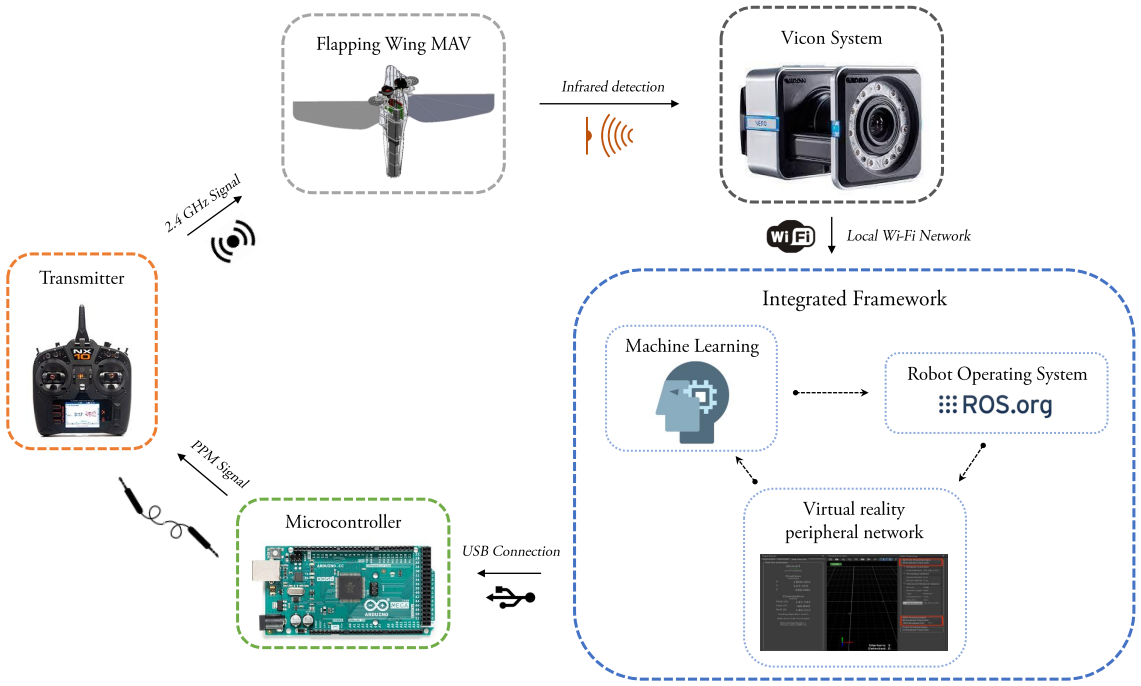


FIGURE 3.1: Closed-loop feedback system of the MAV training setup

In this chapter, the report will understand how information about the flapping wing MAV flows in the form of a closed-loop feedback system; the feedback system allows the reinforcement learning agent to know its current state, take appropriate actions informed by the neural network and then update the following flapping wing MAV's next state. Figure 3.1 clearly shows the flow of information as a cycle.

Beginning at the Flapping Wing MAV, kinematic data recorded by the infrared Vicon cameras. This data passes into our integrated framework, managed by the Robot Operating System (ROS). In the integrated framework, the kinematic data is encoded as position, velocity, and acceleration in SI units according to the cartesian grid illustrated virtual reality peripheral network software. The data continues into the machine learning program (specifically, DDPG). The outputs of the DDPG program feeds into the Arduino microcontroller, which controls the remote transmitter. The transmitter controls the Flapping Wing MAV through a receiver on board, thus completing the information feedback loop.

The rest of the chapter covers each section of the MAV training setup in depth.

3.1 MAV setup

The following design considerations must inform the design of the training setup of the Flapping Wing MAV:

- Training setup should not restrict the ability to train the MAV.
- Training setup should not restrict the flight of the MAV.
- Training setup should capture as much helpful information about the MAV as possible.
- Safety in training as repairing the MAV is a difficult task.

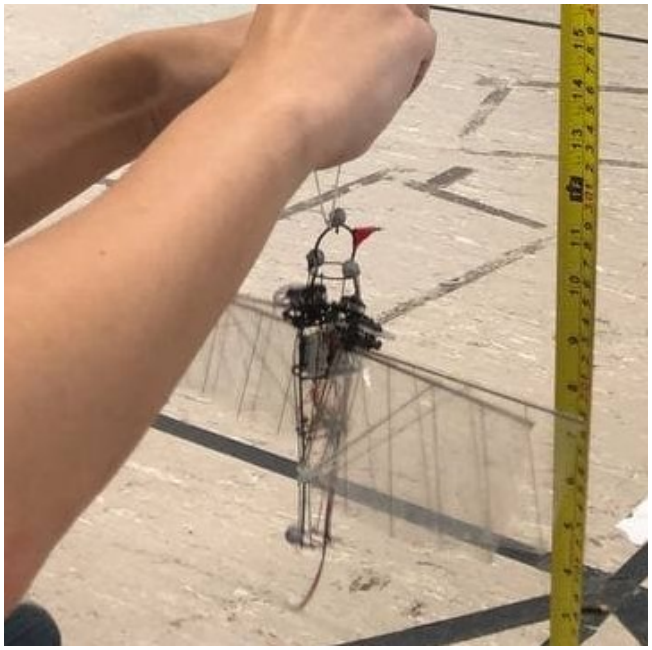


FIGURE 3.2: String and Vicon balls attached to the Flapping Wing MAV

We decided that training the Flapping Wing MAV in a Vicon room with the aid of a hanging string best conforms to the design considerations. We chose to use the Vicon room as a Vicon system is the most effective at recording positional data of the MAV in real-time, with minimal hindrance to the flight dynamics of the MAV. For the Vicon system to capture positional data, small reflective balls or "Vicon balls" must be placed around the MAV. These Vicon balls are incredibly light (less than 1 gram in total) and relatively small. The light weight of the Vicon balls allows the MAV to fly with reduced sluggishness, and their small size decrease aerodynamic drag on the MAV. By recording flight position using external Vicon cameras as opposed to onboard sensors, we prevent any errors caused by odometry drift.

A string was used to hang the Flapping Wing MAV from the ceiling, preventing the MAV from taking severe damage if it drops out of the air while in flight. The length of the string is slightly shorter than the ceiling's height from the floor, allowing the MAV to hang just above the ground at its lowest point. The string was chosen to ensure the MAV's safety as opposed to other methods like using cushions to break the fall of the MAV as the impact of the drop transfers into the MAV frame. The MAV frame

made of thick carbon fibres and spars has a higher yield point than weaker structures like the wings. Hence, it is less likely that the MAV is damaged when dropped.

There are a few disadvantages to using the string to break the fall. Firstly, the MAV has to carry the string in flight, increasing its weight and sluggishness in flight. Secondly, the string restricts the domain of flight to a hemisphere with a radius equal to the length of the string centred on the point it connects to the ceiling. This domain restriction means that the machine learning algorithm has less freedom to explore and learn how the MAV performs in flight as it has to give up control when the MAV approaches the edge of the hemisphere. Nonetheless, we decided that the safety of the MAV outweighs these disadvantages as repairing the MAV is difficult and time-consuming due to the small parts involved. Furthermore, any permanent damage to the MAV could change the flight dynamics, rendering previously trained machine learning models less effective as these models would be working on a different flight system.

3.2 Vicon system

The Vicon room is a large open room with infrared motion capture cameras placed all around the edges of the room. These cameras emit infrared throughout the room. Reflective Vicon balls placed on the Flapping Wing MAV reflect the infrared rays into the cameras, allowing the Vicon system to capture the MAV's motion. The Vicon balls are made of reflective tape taped around a hollow 3-D printed ball, and at least three Vicon balls should be placed around the MAV in a non-symmetric manner. Finally, Wi-Fi is used to transfer the motion data to the Virtual Reality Peripheral Network (VRPN) in our integrated framework.

One obvious disadvantage of using the Vicon system is that the system restricts flight to the Vicon room. The benefits of an AI-controlled Flapping Wing MAV include autonomous navigation and obstacle detection in the real world. While the Vicon system prevents the realisation of such benefits, we decided that the Vicon system allows us to simplify the difficulties of training the MAV in a controlled environment. Future works will include training a fully robust model in the Vicon system before moving on to more "noisy" localisation methods such as inertial sensors and onboard sensors.

3.3 Integrated framework - ROS

To integrate the machine learning program into the Flapping Wing training setup, we chose Robot Operating System (ROS) to facilitate data transfer in and out of our program. ROS enables us to easily receive and manipulate the flapping wing MAV's motion data and send the data as inputs for the machine learning program. The output of the agent is also easily encoded into data that ROS can send to microcontrollers.

Another reason we used ROS as our software framework is because ROS follows a modular philosophy of "complexity via composition". This philosophy provides two advantages. Firstly, it allows each team member to work on different sections of the program without the difficulties of integrating each part together. Secondly, and more crucially, we can swap between different algorithms and training methods by simply redirecting information flow. This advantage improves our program's readability

and modularity, reducing the complicatedness of understanding the lines of code. The following section will briefly introduce terms commonly used in ROS.

Nodes: ROS nodes are a running instance of a ROS program. Nodes can subscribe to data, manipulate them, and finally publish them for other ROS nodes. In our project, we will use four nodes, shown in Figure 3.3 below. Each node has a callback function that runs whenever a message from a subscribed topic enters a node.

Topics: Nodes publish and subscribe on ROS topics. Topics connect nodes and allow data to pass from one node to another in the form of messages.

Messages: ROS messages are data structures that hold data so that different nodes can communicate.

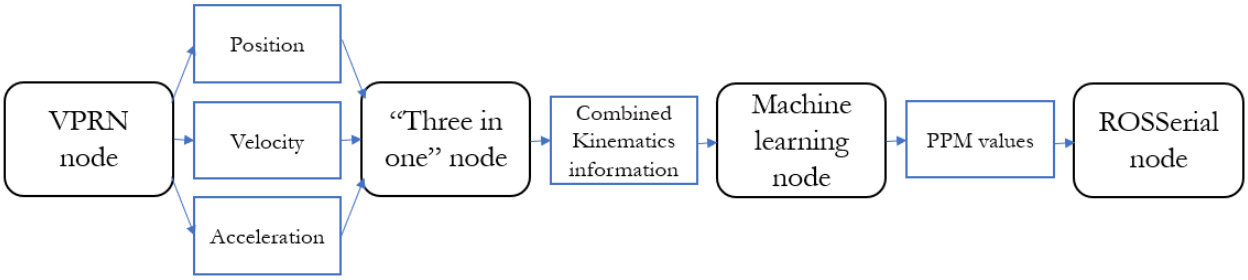


FIGURE 3.3: Nodes and topics of the training setup

3.3.1 VRPN

The VPRN node encodes the Flapping Wing MAV's motion data and sends the data into our integrated framework via Wi-Fi, as explained previously. We have subscribed to three topics for our setup: each sends position, velocity, and acceleration data. The names of the messages are *PoseStamped*, *TwistStamped*, and *AccelStamped*, respectively. These three messages are part of the geometry messages library. All three messages contain three different information: the header, linear motion data (position or velocity or acceleration), and angular motion data.

The header contains meta-information about the message. Importantly, it contains the timestamp of when the message is published. The time is essential for syncing messages together, which will be explained below.

The linear motion data is encoded with three numbers, x, y, z, each specifying position, velocity and direction in each cartesian axis respectively. More specifically, *PoseStamped* describes x, y, z, *TwistStamped* describes $\dot{x}, \dot{y}, \dot{z}$, *AccelStamped* describes $\ddot{x}, \ddot{y}, \ddot{z}$.

The angular motion data is encoded differently for position and for velocity and acceleration. Quaternions are used for angular position, q_x, q_y, q_z, q_w while three numbers x, y, z describes the Euler angles for angular velocity and angular acceleration. More specifically, *PoseStamped* describes q_x, q_y, q_z, q_w, *TwistStamped* describes $\dot{\theta}, \dot{\phi}, \dot{\psi}$, *AccelStamped* describes $\ddot{\theta}, \ddot{\phi}, \ddot{\psi}$.

All three messages are transported through different topics for each message type, and our "three-in-one" node subscribes to the topics. This node uses an approximate time algorithm to read each

message's time stamp from their headers, running the node's callback function only when it has received three messages (position, velocity and acceleration) with roughly the same timestamp. The approximate time algorithm ensures that all kinematic data refer to the MAV's state simultaneously. This node then publishes all 19 values (3 linear positions, 4 angular positions, 3 linear velocity, 3 angular velocity, 3 linear acceleration, and 3 angular acceleration values) in a custom message *KinematicStamped*.

3.3.2 Machine learning

KinematicStamped is subscribed by our machine learning node. The node takes in all 19 values as inputs and 4 pulse position modulation (PPM) values. These PPM values correspond to the signal used by the transmitter, with values from 1000 to 2000. These 23 values are the input for the DDPG. The details of the Deep Reinforcement Learning algorithm can be found in Chapter 2 and Chapter 4. The machine learning node outputs are 4 PPM signals, which we will send to the transmitter through the microcontroller. The PPM signal data is packaged in a message called *ppmchnls*. Lastly, the node will publish the message to a topic that the Arduino microcontroller subscribes.

3.3.3 ROSSerial

ROSSerial is a protocol for wrapping ROS messages so that messages can be sent over a serial port. Specifically, we use the library *rosserial_arduino*, allowing us to run ROS on an Arduino. ROSSerial allows the Arduino to read the *ppmchnls* message.

3.4 Arduino Microcontroller

On the Arduino, a ROS node subscribes to a topic publishing the *ppmchnls* message. In the message, it receives 4 PPM values corresponding to channels one to four on the transmitter. To convert the PPM value ranging from 1000 to 2000 to an actual PPM value, we used a library called *PPMEncoder*. The PPM signal is sent to the transmitter through a mono AUX cable.

The Arduino code is structured in three parts: the setup, the loop and the callback. The setup starts the MAV in its initial state. We have chosen the initial state as wings flapping equally, and the linear actuators centred such that the MAV will be at a neutral attitude and altitude (hovering in place). These values are empirically chosen. While a hovering state is near impossible without an autopilot, our goal is to allow the initial state to be as close to a hovering state as possible. The setup also starts up the ROS node.

The looping part of the Arduino subscribes to the topic from the machine learning node and takes in messages containing the PPM values that the DDPG outputs. When a message enters the node, the callback function is activated. The callback function encodes the PPM value as a signal and sends the signal through an output pin.

3.5 Transmitter

The transmitter used in this project is a Spektrum NX10. The transmitter is set to instructor mode to transmit the PPM signal as 2.4 GHz EM waves. In instructor mode, the transmitter accepts the PPM signal through the AUX cable and emits the EM waves through its antenna. The EM signals are then picked up by the receiver on the Flapping Wing MAV, thereby completing the feedback loop.

4 Reinforcement Learning Cast

4.1 Agent & Environment

Understanding the entire hardware & software training setup of the MAV is crucial to the next important step, the reinforcement learning cast. Now that we have a physical understanding of the setup and the goals we want to achieve, we can cast this knowledge into the MDP framework as defined in section 2.1. This is very crucial as the mathematics of reinforcement learning can only justify the setup if and only if the cast is done properly.

We begin by defining the agent & environment. The agent here is **strictly** defined as the code running on the machine learning script of the integrated system running on the laptop. All of the code running is deterministic (assuming no bugs) & the agent in the code is making all the decisions. This may seem counter-intuitive, as that means that the environment not only consists the string attached to the MAV or the physical space in which the MAV can move, but **the MAV itself**. Although the MAV is the robot learning to fly, we must recall that the training objective is not for the MAV itself to fly, but for an agent to control and fly the MAV. As far as the agent is concerned, the MAV is merely but a vessel that carries out its instructions. Even if the agent was present as a companion computer on the MAV, the computer chip where the processing occurs would be considered the agent and all other mechanical, electric and flapping parts are still considered as the environment. Such a strict definition on agent & environment is necessary as the MAV itself is fully subjective to non-deterministic behaviour. Recall the need for a state transition probability, $f(S', A, S)$, as illustrated in Figure 2.2. Any component of the full system that cannot be confidently deterministic must be part of the environment which affects $f(S', A, S)$. The actual MAV does not have a deterministic behaviour. Components can break and fail in flight, the battery voltage may be too low to sustain the same level of thrust, the motors may reduce its effectiveness over time and many more unplanned events can happen. If the wings fail, the S' achieved from the exact same S & A can be different if the wings are intact. This is mathematically factored in as part of the state transition probability that the MAV must learn.

Naturally, a good engineer must do all it takes to reduce the chances of unexpected events, but it is not possible to consider the MAV a purely deterministic system. One may argue that the state being recorded from the Vicon system is that of the actual MAV and not the code, and the state describes the agent's situation, so why is the MAV not the agent? Although a valid point, we believe that a state does not precisely describe the situation of the agent, rather it describes the situation of the thing the agent is trying to control. In this case, the latter is rightfully the MAV. In most reinforcement learning problems, the agent and the thing the agent is trying to control is usually the same thing, so such distinctions are not usually necessary. A unique real-time mechanical problem like the one in

this project brings out the need for the distinction on what a state truly describes. Hence, we define the MAV as part of the environment.

4.2 State

The state for the MAV can be defined by simply 19 float variables. They are the position x, y, z, the orientation in quaternions, Q_x, Q_y, Q_z, Q_w, the linear and angular velocities, $V_x, V_y, V_z, \omega_x, \omega_y, \omega_z$, and the linear and angular accelerations, $A_x, A_y, A_z, \alpha_x, \alpha_y, \alpha_z$. Deciding to use these 19 values as the state is pretty straightforward. The goal of controlling the MAV to fly is a mechanical objective, so it makes intuitive sense to use physical characteristics to define the state of the MAV. As MAV flight is very sensitive to its physical parameters, we believe that velocity & acceleration information is valuable in describing the MAV completely. Moreover, these are the only 19 values that we can get from the Vicon system. One could consider doing some feature engineering to reduce the number of state variables, however 19 inputs into a neural network is comparatively little and takes minimal computational resources to handle.

One advantage of taking the physical characteristics of a rigid body is that the characteristics are all we need to sufficiently describe the state of the MAV. No information of the state in the previous time-steps make a difference to the state of the MAV currently and hence the future state and rewards of the MAV are not conditional on the past. Hence we can confidently claim that the state of the MAV follows the Markovian assumption. Therefore, there is no violation to casting this system as a MDP.

4.3 Actions

Defining the actions cast is the most challenging and interesting. The PPM signal that the Arduino must send to the transmitter is within a range of 1100μs and 1900μs. 1100μs refers to the lowest value of a channel and 1900μs is the highest. Hence the latter would mean maximum throttle on the motor channels. Figure 4.1 shows the electric diagram of the MAV.

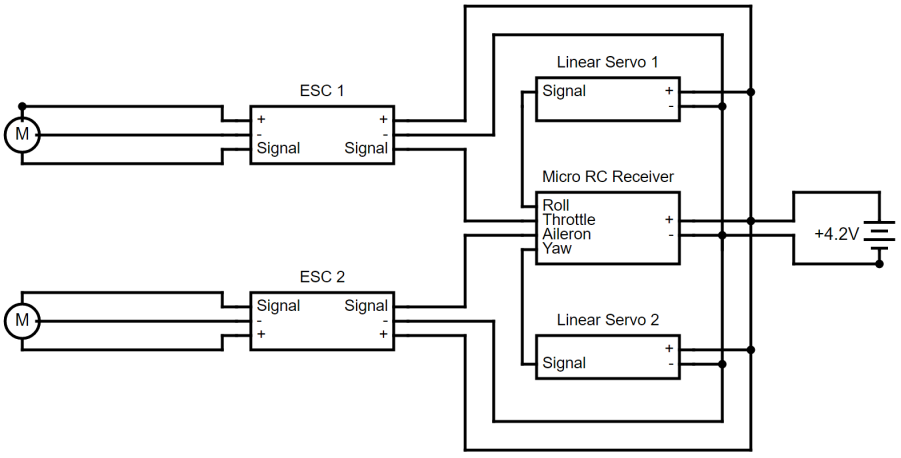


FIGURE 4.1: Electric Diagram of the MAV. 4 Channel receiver is used.

The diagram shows that the receiver accepts 4 channels and each one these 4 channels are connected to an actuator (2 motors & 2 linear servos). Hence the Arduino must output four PPM signals within the range of 1100μs and 1900μs. Therefore the first idea is to have an action space of size 4, that each select a value between 1100μs and 1900μ. The output of each neuron in DDPG is an output from a hyperbolic tangent function, η, such that $\eta \in [-1, 1]$. The conversion from η_1 to the PPM_1 (where the subscript 1 refers to the first channel) is displayed in Equation 4.1.

$$PPM_1 = \eta_1 \frac{1900\mu s - 1100\mu s}{2} + \frac{1900\mu s + 1100\mu s}{2} \tag{4.1}$$

This initially seemed like a good idea, however we observed a big drawback when testing the training setup with the actual MAV. At the start of training, the neural networks have not yet been trained, hence the agent often chooses rubbish PPM values for the MAV. The agent can command the MAV to go full throttle in one time step, and then no throttle in the net time step and then back to full throttle in the subsequent time step. This is possible as the agent can pick any value between 1100μs and 1900μ in each time-step. This put incredible amounts of stress on the motor which was applying a lot of torque on the gearbox due to slowing down and speeding up quickly. This meant that there was a lot of shear force on the connecting components of the gearbox, eventually leading to the failure of one of them. Figure 4.2 illustrates this very broken component.

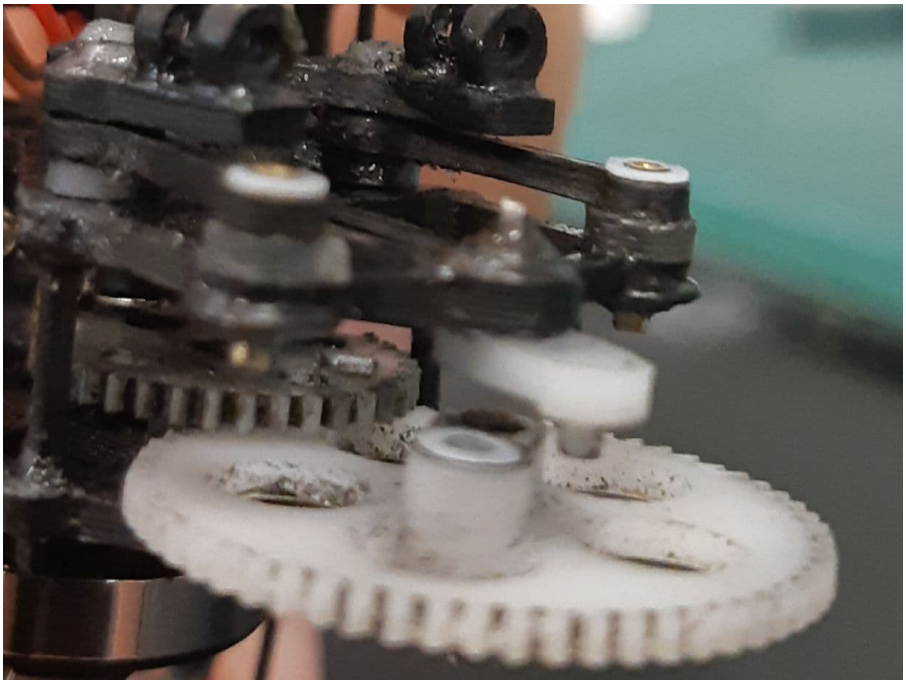


FIGURE 4.2: Damaged Gearbox component from high shear force after agent made rapid changes to motor speeds

Another aspect of reinforcement learning that one can take for granted is the assumption that the thing the agent is controlling has had enough time to complete the full action before the next state is recorded and the next action is sent out. In real-time systems, this assumption does not always hold as the S, A, R, S', A' may have overlap in time. On the software side, the rate at which states are

recorded is controlled at an optimal rate of 5Hz. Making it too slow can be detrimental as well, as the MAV may have finished its actuation and any new changes in state may not be attributed to the action change. This is only relevant for the linear servo as it is a single-use actuation unlike the motor which is constant actuating above a PPM of 1100μs. Right after the linear servo rotates the wing, the state should be immediately captured so that the agent accurately determines the $S \rightarrow S'$ transition as a result of the linear servo change. Likewise, the next action can be sent out at this exact same time. The issue with having an agent that can dictate rapidly different consecutive actions is that the MAV may not even have time to finish actuating, precisely the issue just discussed. Hence using this approach for actions was deemed unsuitable.

Instead of mapping the DDPG output to the absolute PPM values, we define the output to be proportional to the change in PPM instead of PPM itself. We call this ΔPPM. Let us define another constant ζ that is equal the maximum allowable change in PPM between two time-steps. This value was not tuned for this project, but we can assume a default value of 200μs. ΔPPM_1 is simply defined as in Equation 4.2.

$$\Delta PPM_1 \mu s = \zeta \eta_1 \mu s \tag{4.2}$$

Then the value of PPM_1 is updated to $PPM_1 + \Delta PPM_1$. The updated PPM_1 is clipped to ensure it lies in the range between 1100μs and 1900μ. This revamped action cast performed on the MAV was observed to be a lot less mechanically stressful on the gearbox and motor.

4.4 Rewards

The rewards structure of any reinforcement learning project can be heavily customised by the human programmer. This is the one area in which there is the most flexibility, however it is arguably the most important. The way in which the reward structure is designed is crucial to the agent's perceived goal of the training. The incentives and punishments must be clearly drawn out so the agent learns to control the MAV exactly in the way we want it to. An unclear goal or a mix of multiple goals can end up confusing the agent.

As we want the agent to control the MAV to fly, we provide a positive reward for upward velocity and a negative reward if it has a downward velocity. Any additional rewards are purely dependent on the training methodology applied. The following chapter will detail them and give more insights on the reward structure to implement.

5 MAV Training Goals

5.1 Goal 1: Fly with string

The goal of wanting the agent to "control and fly the MAV" is very vague. In reinforcement learning, there is no room for ambiguity. Any training objective must be clearly laid out. In method 1, we propose the goal of the agent controlling the MAV to use the string to gain altitude. By using the string's tension, the MAV can achieve circular motion and use a simple increase in throttle to gain altitiude. This scenario is easy for the MAV to achieve as it does not require the MAV to be stable in flight to do this. It mostly just has to apply constant full throttle. Figure 5.1 shows an instance of when the MAV is tugging on the string to gain altitude while moving around the room in a circular motion.

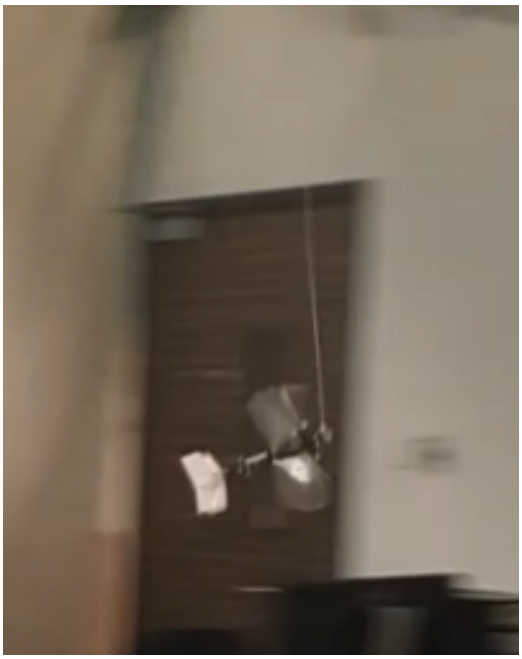


FIGURE 5.1: MAV relying on the string's tension to gain altitude

Indeed, this is not a very useful application of the MAV as it does not achieve true flight. Nonetheless, this training methodology is a good litmus test of the software written to train the agent to control and fly the MAV. This is an achievable task and if the agent can learn to accomplish it, it means that the full MAV training setup is functional and no serious code bugs are in play. Hence, we recommend this as the first training goal to follow. There are no additional reward functions needed for this goal as it is a simple task of going up.

5.2 Goal 2: Fly with and without string

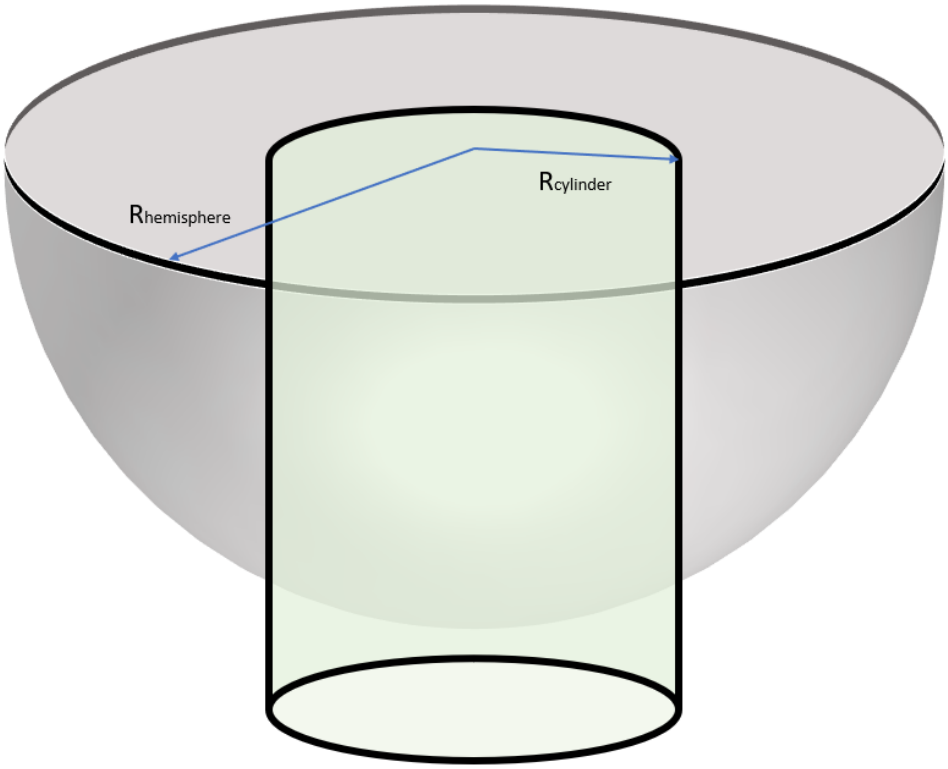


FIGURE 5.2: Hemisphere of physically possible flight domain and cylinder of permissible flight domain

One way to achieve an upward motion using reinforcement learning is to restrict the flight of the MAV to a smaller cylindrical domain inside the hemisphere enforced by the string (See Figure 5.2). We allow the MAV to use the string's tension where the surface of the hemisphere and the cylinder's domain intersects to a certain height, where the surface of the cylinder and hemisphere intersects. After a fixed height, we place penalties on the agent when the MAV leaves the cylinder's domain. This scenario allows the MAV to explore and gather speed at the bottom of the cylinder, and when it has sufficient speed, learn to fly without the tension of the string. The reward for altitude climb is given only after this height.

This method circumvents the problem of Section 5.1, and the MAV will learn to fly upwards without the string. However, we identified one serious problem: relying on the string at one region of space and without the string at another will confuse the agent as the environment "physics" changes. This problem could cause a bias in the neural network, where at lower altitudes, it assumes there is always an upwards tension to reduce the MAV's weight.

5.3 Goal 3: Fly without string

The last method of achieving an upward motion is simply to release the MAV from a reasonable altitude with no tension in the string and allow the MAV to attempt to fly. The string, in this context, serves purely as a safety net for the bird to prevent and damages from upon crashing. During training, the episode is immediately terminated when the bird leverages on the string's tension at any given point of time in the flight or leaves from an small arbitrarily specified range best visualised by the diameter of the cone shown in Figure 5.2. An immediate advantage of the stated training method would be that the environment "physics" remains constant throughout the training process unlike the method stated in Section 5.2. In addition, the reason for such strict terminating conditions is to ensure that only desirable experiences of the MAV are stored in the replay buffer, where the desirability of the experience is determined by the region in the state space that the MAV is likely to be in when is attempting to fly upwards (e.g. orientated upright with MAV head facing the ceiling). The experiences of the MAV going from a desirable state to another desirable state (e.g. state with increase height and upright orientation) is naturally important. In addition, the experiences of the MAV going from a desirable state to a undesirable state (e.g. pitching downwards) is also important as the model needs to learn not to select the combination of actions at the desirable state that led to to a non-desirable state through reduced rewards from the reward function. However, experiences where the MAV is struggling to recover from a undesirable state to another undesirable state is not valuable given that it is a region in the state space where the MAV would not be if it was achieving its goal of upward flight, hence only serving as noise that hinders learning. Hence, the stated training method maximises the collection of quality data for training to achieve the desired goal, hence the most sample efficient and effective method theoretically.

6 Conclusion

6.1 Summary

In this project, the team has developed the integrated framework from scratch and integrated the entire system architecture for the closed-loop feedback system of the MAV training setup as illustrated in Figure 3.1. Firstly, the team has assessed several deep reinforcement learning algorithms on their suitability in the context of the MAV training and eventually settled for the state-of-the-art policy gradient reinforcement learning algorithm DDPG. The entire code for DDPG was then developed from scratch and interfaced with the with the Vicon system and the Arduino microcontroller through the ROS program, of which is also developed from scratch. Thereafter, the team integrated the Arduino microcontroller with the transmitter and naturally from the transmitter to the MAV as well via an on-board receiver. The MAV was also equipped with Vicon balls for the Vicon system to detect its state, thereby completing the closed-loop feedback system of the MAV training. After developing training setup, a reinforcement learning cast was performed to fully contextualise entire problem of achieving flight for the MAV in well defined reinforcement learning terms. In particular, the methodology of achieving various well defined training goals are detailed and examined for their merits and disadvantages in the context of reinforcement learning. In essence, the team has developed all the necessary systems required to for any individual to train the MAV to fly.

6.2 Further studies

Naturally, further studies would definitely involve the actual training of the MAV itself in the system developed. Only from actual training can the team evaluate the performance of the reinforcement learning algorithm and develop the necessary changes to the machine learning aspects in order to achieve the goal of MAV flight in the most efficient manner possible. The necessary changes could involve hyperparameter tuning (e.g. reward structure, network architecture) for DDPG algorithm or even the usage of another reinforcement learning algorithm. Furthermore, it is important to note that the MAV used currently is inherently unstable and it does not have any autopilot nor any control surfaces on its tail to stabilise it. As a result, manual flight with it is practically impossible as no human have the sufficient reaction time and capacity to react accordingly to the MAV's instability. However, should another MAV that is inherently stable and can be piloted by a human be used, further studies could involve experimenting with imitation learning, where the human pilot provides the to "correct" inputs to fly the MAV, of which the experiences are stored in a replay buffer from which the model can learn from. It is likely that such a method would be significantly more efficient in training the model as the quality of the experience generated from an experienced pilot is vastly superior to

that of a randomly initialised reinforcement learning algorithm attempting to learn from exploration in the early stages of the training. Lastly, should the task of upward flight be achieved, further studies could involve changing the goal to fly the MAV to a specified waypoint from the original location of the MAV and hover at that location.

Bibliography

[1] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.