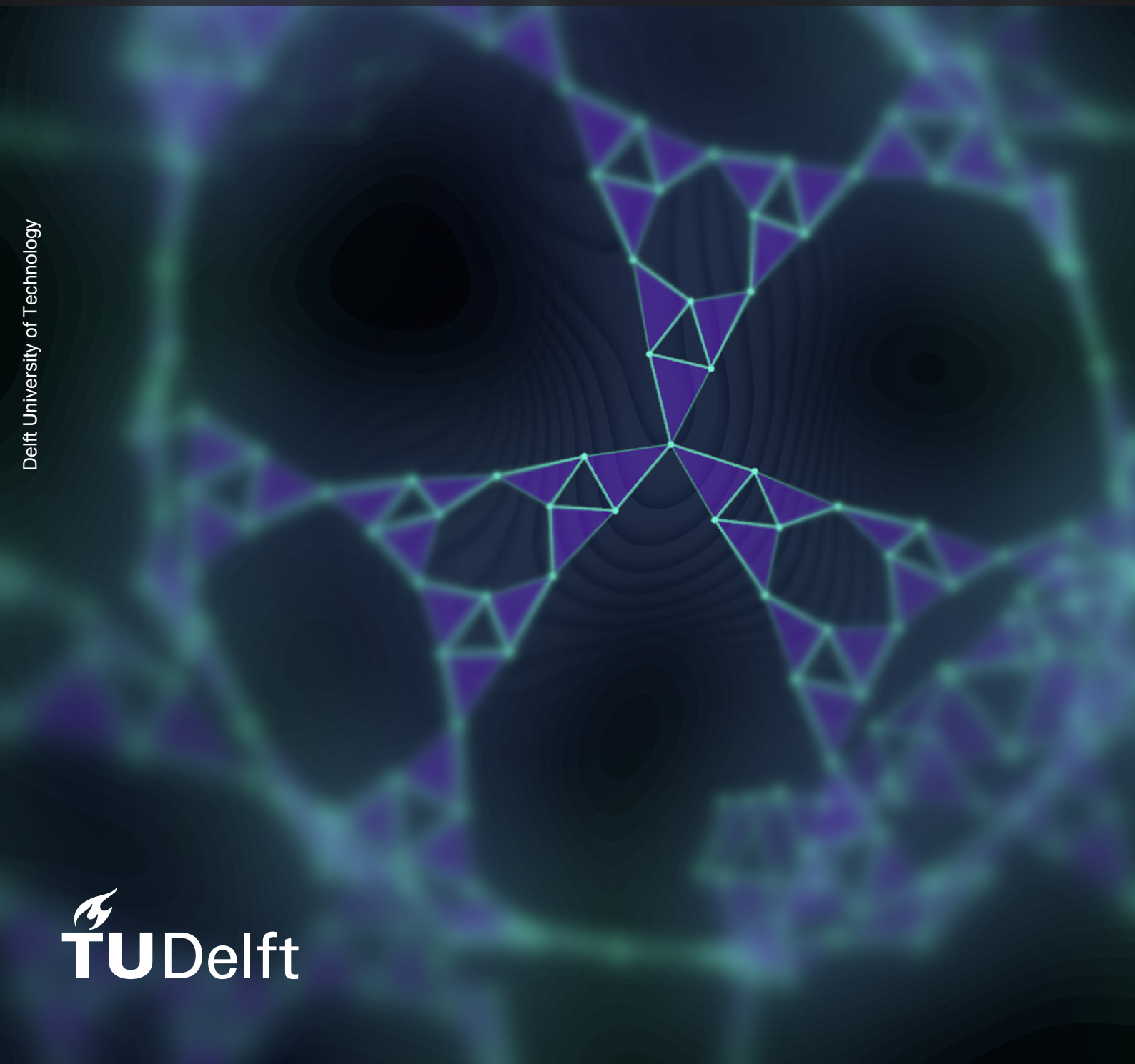
Reinforcement Learning for Flight Control

Advancing Intelligent Fault Tolerant Control

Wing Chan



Delft University of Technology

TU Delft

Reinforcement Learning for Flight Control

Advancing Intelligent Fault Tolerant Control

by

Wing Chan

Supervisor: Prof. Dr. Erik-Jan van Kampen
Institution: Control & Simulations, Faculty of Aerospace Engineering, TU Delft
Place: Delft, The Netherlands

TU Delft

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

Latin Symbols

A_t	Action at t	$[-]$
a	Action variable	$[-]$
S_t	MDP State at t	$[-]$
s	MDP State variable	$[-]$
R_t	Reward at t	$[-]$
r	Reward variable	$[-]$
$\mathcal{T}$	Trajectory	$[-]$
T	Episode final timestep/episode termination time	$[-, \mathrm{s}]$
$p()$	Probability distribution	$[-]$
$\mathbb{E}()$	Expectation operator	$[-]$
G_t	Return at t	$[-]$
$f()$	State transition function	$[-]$
$g()$	Input/action transition function	$[-]$
Q	LQR state cost parameter matrix	$[-]$
R	LQR action cost parameter matrix	$[-]$
$\mathcal{E}$	Eligibility trace vector	$[-]$

Greek Symbols

$\pi()$	Policy function	$[-]$
$v_\pi()$	State-value function under policy π	$[-]$
$q_\pi()$	Action-value function under policy π	$[-]$
α	Learning step size	$[-]$
γ	Reward discount rate	$[-]$
θ	Function approximator parameter vector	$[-]$
η	Multi-step learning parameter	$[-]$

Nomenclature

Abbreviations

MDP	Markov Decision Process
HDP	Heuristic Dynamic Programming
DHP	Dual Heuristic Programming
GDHP	Globalized Dual Heuristic Programming
AD	Action-Dependent
ACD	Actor-Critic Design
ADP	Approximate Dynamic Programming
VI	Value Iteration
PI	Policy Iteration
LQR	Linear Quadratic Regulator
MC	Monte Carlo
TD	Temporal Difference
HJB	Hamilton-Jacobi-Bellman
RLS	Recursive Least Squares
TD	Temporal Difference
IHDP	Incremental Heuristic Dual Programming
IDHP	Incremental Dual Heuristic Programming
MIDHP	Multi-step Incremental Dual Heuristic Programming
DRL	Deep Reinforcement Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
DQN	Deep Q Network
SGD	Stochastic Gradient Descent
TRPO	Trust Region Policy Optimization
PPO	Proximal Policy Optimization
KL	Kullback-Leibler
DPG	Deterministic Policy Gradient
DDPG	Deep Deterministic Policy Gradient
TD3	Twin Delayed Deep Deterministic policy gradient
SAC	Soft Actor Critic
LTI	Linear Time-invariant
DSAC	Distributional SAC
AoA	Angle of Attack
LTV	Linear Time-Varying
IQR	Inter Quartile Range
PID	Proportional Integral Derivative

1 Introduction

1.1. Background and Motivation

Civil aviation is a sector that has a good track record in terms of transport safety [1], this excellent record is nonetheless tarnished by individual accidents that involve high fatality rates. The most fatal category of such accident is known as Loss of Control – Inflight (LOC-I), which groups together accidents that occur due to an aircraft deviating from flight path or aircraft operation outside normal flight envelope; 94% of accidents in this category resulted in fatalities [2]. Fault-tolerant or robust controllers are flight control methods that remain performant and stable despite changes in the dynamic properties of an aircraft or operation in extreme conditions. One of the most exciting areas of research in fault-tolerant controllers is the application of reinforcement learning agents to control an aircraft. This approach can allow for directly optimizing a controller for scenarios that lead to LOC-I accidents [3], or even online adaptation of controller parameters in the face of dynamic behaviour changes [4], all of which have the potential of providing the flight controller with greater safety margins.

The limits to operational improvements do not stop at greater safety, as a higher degree of fault-tolerance will improve the feasibility of fully autonomous systems. The benefits of this range from the possibility to deploy pilot-less cockpits, to enhancing swarm-based drone systems for purposes such as offshore inspection or large-scale search and rescue operations.

The potential impact of reinforcement learning on the control engineering community should be noted as well. Current industry practices for controller synthesis for aerospace engineering are predominantly done by combining Proportional Integral Derivative (PID) control laws with gain scheduling [5], where a different set of PID gains are designed for each discrete point in the flight envelope. While this approach has been proven to work and serves the aviation industry well, it is nonetheless tedious, time-consuming, and thus expensive to perform, requiring the expertise of dedicated control engineers. Furthermore, while such controllers provide guarantees for the automatic flight control system's performance within all considered flight conditions, such guarantees do not extend to portions of the flight envelope not covered by or between the selected flight conditions. This can happen in the case of faults or extreme weather conditions.

In addition to flight safety and control design efforts, there is also the issue of sustainability. Based on 2018 data the aviation industry is responsible for approximately 2.4% of global anthropogenic greenhouse gas emissions [6]. Despite aviation traffic volumes falling drastically as a result of the COVID-19 pandemic, the recovery of air travel has been rapid and the number of flights is expected to recover soon, ultimately resulting in the aviation sector contributing 0.1 degrees Celcius of warming by 2050 [7]. To alleviate the severity of climate change, the aviation industry must therefore become carbon neutral, which will require innovation in all aspects related to aviation. On this front, novel aircraft design concepts such as the Flying-V promise fuel efficiency improvements over current tube-and-wing designs [8], and thus increasing the technological readiness level of the Flying-V is of special interest for sustainability. This maturity will require dedicated efforts toward the design of flight control systems, which traditionally involves dedicated expert knowledge of control theory and time-intensive design campaigns [5]. Furthermore, the guarantees that traditional linear control laws promise would be hampered by the lack of exact system specifications which is present in innovative aircraft designs. By their adaptiveness, reinforcement learning based flight controllers have the potential to be agnostic to what aircraft is being controlled and require less expert involvement in controller design, freeing up

expertise for tackling other pressing design issues. Therefore, reinforcement learning stands for an exciting means of improving the automatic flight control system design procedure, especially in the case of innovative aircraft designs such as the Flying-V.

1.2. Research Goal

Driven by the desire to contribute to the state of the art of reinforcement learning, and motivated by the potential benefits which reinforcement learning based controllers have for the aviation industry, the present research thesis is commenced.

To define the scope of this research, a research objective stating the high-level aim of the thesis, as well as a set of research questions which defines the direction of research more precisely, are stated:

Research Objective

To improve the fault-tolerance of reinforcement learning based flight controllers by advancing the state of the art, through researching novel augmentations to reinforcement learning algorithms.

Research Questions

1. What promising reinforcement learning algorithm for fault-tolerance and tracking performance should be further studied in the present research?
   (a) What reinforcement learning algorithms are considered to be state-of-the-art?
   (b) How is fault-tolerance defined and tested in past research?
   (c) Which algorithms have been shown to provide the best fault-tolerance?
   (d) What reference tracking performance have these algorithms shown in past research?
   (e) What promising augmentations to reinforcement learning algorithms can be made and experimented with?
2. How can the identified algorithm be applied to control the PH-LAB research aircraft?
   (a) How can the identified augmentations be made to the reinforcement learning algorithm studied?
   (b) How should the flight control system be structured?
   (c) What are the variables defining the MDP in the case of controlling the PH-LAB?
3. How does the developed flight controller perform during nominal flight and in the presence of faults?
   (a) What flight scenarios should be designed to test the proposed controller's nominal performance and fault-tolerance?
   (b) How should a controller's nominal performance and fault-tolerance be measured?
   (c) What is the implemented controller's nominal performance and fault-tolerance?
   (d) How do the proposed augmentations affect the characteristics of the baseline reinforcement learning algorithm?
   (e) How well does the proposed controller's nominal performance and fault-tolerance compare to a traditional PID controller?

1.3. Thesis Outline

With the motivation and the goals of the research presented, the remainder of the thesis report can be outlined. The present introduction chapter is first succeeded by Chapter 2, which contains the results of the literature study conducted as part of the research goals, and provides the necessary theoretical background to the technical research work to be conducted. Following this, a chosen reinforcement learning algorithm is to be studied further, implemented, and tested on several conditions. The results of this work in addition to the background of this work are put forward in Chapter 3.

2

Literature Study

The thesis project will commence with a study of literature, this is intended to introduce the main concepts that will be used as the foundation of the methodologies used, as well as supplement the discussion of any results gathered. Literature studies are conducted in all fields related to the research project; for the present research, the main relevant research fields are that of reinforcement learning -constituted largely of approximate dynamic programming and deep reinforcement learning research-, and the field at the intersection of reinforcement learning with flight control.

This chapter is laid out as follows. Section 2.1 introduces the basic concepts in order to lay the theoretical understanding that underlines most reinforcement learning research. This is followed by two sections that discuss the two main research fields dominant in reinforcement learning for control applications: dynamic programming discussed in Section 2.2, and deep reinforcement learning discussed in Section 2.3. These two sections will start with touching on the earlier and more fundamental algorithms, shifting focus progressively to the respective state of the art. The problem of applying reinforcement learning to flight control is then discussed in Section 2.4, which will present various research efforts towards this goal and interesting directions for further study.

2.1. Reinforcement Learning Foundations

The basic idea of reinforcement learning is to have some agent associate rewards with actions that help realize a goal and promote the agent to take more of such actions by asking it to maximize the reward. The agent is not told what the goal is explicitly, its only interface with the world around it is through executing these actions, and observing that it has transitioned into some state and received some reward. Reinforcement learning can be thought of as a way in which theorists have attempted to codify and formulate algorithms for, the universal experience of learning through trial and error. Much like how a child learns to walk, or a dog learns to sit, it is through reinforcement learning that machines can learn how to perform tasks that might not be easily programmed. Through this codification, computer programs have been made that demonstrated impressive levels of learning; for example, programs can learn through reinforcement learning to play various high-dimensional board games to a level of expertise surpassing any living player [9], and a robotic arm can learn hand dexterity and mimic the hand movements of a human [10].

Understanding how the reinforcement learning algorithms work behind such examples and how they may be applied to flight control will require understanding the foundations first, thus this section will introduce the basic terminologies and ideas used to build these algorithms.

2.1.1. Markov Decision Process

The MDP is the mathematical framework that is used to model sequential decision processes such as how an agent interacts with an environment, and it is what contextualizes all the ideas in reinforcement learning, for instance, the basic notion that an agent performs an action and receives a reward.

In such a framework, there exist two entities: the agent and the environment, and information flows from one entity to another to model making decisions and their resulting consequences. In reinforcement learning the agent is sometimes also called the learner, it selects an action A_t which gets fed to the environment, and the environment will provide the corresponding state S_{t+1} which the agent has

transitioned to as a result of action A_t and the reward associated to that state transition R_{t+1}, the agent can subsequently use S_{t+1} and R_{t+1} to decide on the next time step's action A_{t+1}. A graphical depiction of this agent-environment interface in the MDP is shown in Figure 2.1.

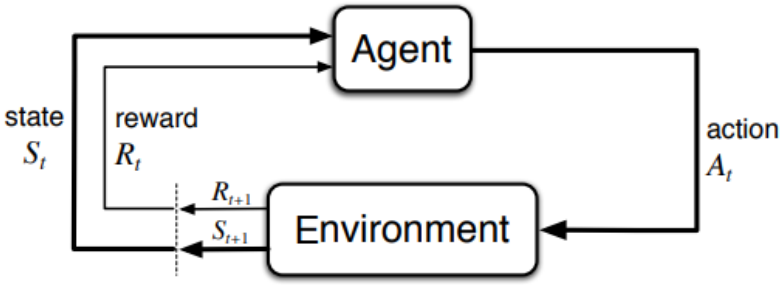


Figure 2.1: Flow diagram of the agent-environment interaction central to the MDP

This time trace of an agent-environment interaction is recorded in a so-called *trajectory* $\mathcal{T}$, which is a chain of state-action-reward-next state values for the entire duration of the decision process:

$$\mathcal{T} = S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T, S_T, A_T \tag{2.1}$$

The state-action-reward-next state of one timestep is commonly collected into one tuple of variables, which can be referred to as a *transition tuple* or *experiences*:

$$(S_t, A_t, R_{t+1}, S_{t+1}) \tag{2.2}$$

One central component of MDPs is the dynamics of the environment. In the simpler case of finite and discrete processes, the dynamics of the environment can be considered to be a discrete conditional probability distribution $p(s', r|s, a)$ as shown in Equation 2.3. which returns the probability of transitioning to a state s' and obtaining a reward r given that the agent observed a previous state s and executed an action a.

$$p(s', r|s, a) \doteq Pr\{St = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.3}$$

From Equation 2.3, it is possible to compute the expected reward for any state-action pairs $r(s, a)$:

$$r(s, a) \doteq \mathbb{E}\{R_t|S_{t-1} = s, A_{t-1} = a\} = \sum_r r \sum_{s'} p(s', r|s, a) \tag{2.4}$$

Modelling the environment dynamics, i.e. the MDP dynamics, can also be done using other modelling methods. For example, state space systems of equations are commonly used when creating control systems, and as such are also used to serve as the dynamics model in the MDPs [11]–[13]. In both cases, one important property that the models possess is the *Markov property*, also referred to as the memoryless property. This property states that to predict the system in the next time step, only information from the current or time step before is necessary, which means that having any information from previous time steps does not influence the outcome of the prediction. This property is captured in Equation 2.5.

$$Pr\{S_t, R_t|S_{t-1}, A_{t-1}\} = Pr\{S_t, R_t|S_1, \ldots, S_{t-1}, A_1, \ldots, A_{t-1}\} \tag{2.5}$$

In reinforcement learning, having the dynamics of the environment possess the Markov property is useful as many algorithms assume that the evolution of the system can be perfectly predicted by only using information from the current time step [14], which should be sufficient for a learner to decide what actions to take in order to enter into a trajectory which maximizes it's rewards.

In the context of flight control, the MDP can be conveniently formulated using state space systems. For example, the action, state, and reward in Figure 2.1 can be considered equivalent to the control vector, the state vector, and an output vector in the state space formulation respectively.

2.1.2. Rewards and Returns

The way in which learners in a reinforcement learning problem gauge their performance is through reward signals, this reward signal is made to be representative of the goal of the reinforcement learning problem.

Reward signals are central to the *reinforcement* in reinforcement learning, as they are made to be associated with states and actions that get the agent closer to the goal, thus incentivizing the learner to repeat more of such actions with the ultimate effect of the agent becoming more proficient in the posed task. Strong parallels can be drawn between an agent in the reinforcement learning context becoming better at obtaining higher reward signals, and that of animal behaviour adapting to receive more desirable stimuli in the context of domestic animal training, a so-called "Law of Effect"[15]. This parallel arises from the strong connections between reinforcement learning as a computer science and mathematical theory, and reinforcement learning as a field of psychology, and shows how ideas in reinforcement learning are often grounded in ideas from nature.

In reinforcement learning nomenclature, a distinction in terminology exists between the reward being received every time step, and the cumulative reward that an agent receives over many time steps. While rewards are received every time step, when they are summed up over time, they are then referred to as the *return*. For example, an episodic return refers to how much cumulative reward an agent has received throughout an episode. Formally, returns are defined by Equation 2.6 as the sum of all future discounted rewards, where the discount factor $\gamma \in [0, 1]$ is introduced which allows for returns to be computed even when a task is not episodic but continuing, i.e. $T = \infty$. Increasing γ from 0 to 1 increases the weight of rewards received later in time, which makes the agent more "far-sighted", the opposite case of decreasing γ causes the agent to be more "myopic".

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.6}$$

The goal of a learner in reinforcement learning is to maximize the return G_t, this return serves as one of the main metrics for evaluating the success of an agent. Moreover, the task of designing appropriate reward signals is one of the most important when posing a reinforcement learning problem. If the desired outcome is not associated explicitly with rewards, an agent would likely not learn to reach such an outcome.

2.1.3. Policy

A policy is what an agent follows to decide on what action to choose. Formally, a policy π is defined as a functional mapping from state s to the probability of choosing an action a, i.e. probability of action a conditioned on state s:

$$\pi(a|s) \doteq Pr\{A_t = a|S_t = s\} \tag{2.7}$$

The specific formulation of $\pi(a|s)$ differs greatly from algorithm to algorithm, in fact, there is no restriction on the form of the probability distribution that $\pi(a|s)$ takes. For instance, it is permissible that $\pi(a|s)$ is deterministic, i.e. if the state is s_1, then action is a_1. The overarching theme is that an agent should follow a policy that will maximize its returns.

2.1.4. Value Function

Value functions predict how much return an agent will receive in expectation if it were to follow a policy π. Two types of value functions are commonly used in reinforcement learning, the state-value function $v_\pi(s)$ and the action-value function $q_\pi(s, a)$. The state-value function is the expected return from being in any single state, mathematically this is defined as:

$$V_\pi(s) \doteq \mathbb{E}_\pi\{G_t|S_t = s\} \tag{2.8}$$

Where $V_\pi(s)$ is the value of state s, and G_t is the return from time t onwards when following the policy π. Notation of the expectation operator is slightly abused to include the π to indicate that the agent is following policy π. The state-value function can be intuitively understood as how worthwhile it is to be in a certain state. The action-value function is defined as the expected return of a particular state-action pair, this can again be mathematically stated as follows:

$$Q_\pi(s,a) \doteq \mathbb{E}_\pi\{G_t|S_t = s, A_t = a\} \tag{2.9}$$

Where $Q_\pi(s,a)$ is the value of taking action a in state s, and G_t is the return from time t onwards when following the policy π. The action-value function can be intuitively understood as how worthwhile it is to take a certain action in a certain state.

2.1.5. Bellman Equation

The equation commonly referred to as *the* Bellman equation is the Bellman equation for the state-value function, presented in Equation 2.10, which is derived starting from the definition of the state-value function Equation 2.8:

$$\begin{aligned}
V_\pi(s) &\doteq \mathbb{E}_\pi\{G_t|S_t = s\} \\
&= \mathbb{E}_\pi\{R_{t+1} + \gamma G_{t+1}|S_t = s\} \\
&= \sum_{r,g_t} p(r, g_t|s)[r + \gamma g_t] \\
&= \sum_{a,s'}\sum_{r,g_t} p(a, s', r, g_t|s)[r + \gamma g_t] \\
&= \sum_{a}\sum_{s',r}\sum_{g_t} \underbrace{p(a|s)}_{=\pi(a|s)}\, p(s', r|s, a)\, \underbrace{p(g_t|s', r, s, a)}_{=p(g_t|s'),\ \text{Markov}}[r + \gamma g_t] \\
&= \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{g_t} p(g_t|s')g_t] \\
&= \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')]
\end{aligned} \tag{2.10}$$

This equation defines an analytical relationship between the value function of successive states, suggesting that the value of a previous state can be calculated once the subsequent state's value function, the environment dynamics, and the followed policy are known.

Notice that the value function in Equation 2.10 is recursively defined. This recursive formulation lends itself readily to dynamic programming techniques, which simply means that Equation 2.10 is adopted as an update equation for estimates of the state value function, where by iteratively applying this update the estimate converges towards the true value. The Bellman equation of Equation 2.10 was first derived by Richard Bellman and is associated with the basic foundations of the field of dynamic programming [16].

There also exists a Bellman equation for the action-value function, which is presented in Equation 2.11.

$$Q_\pi(s,a) \doteq \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')] \tag{2.11}$$

2.1.6. Distinguishing Algorithm Characteristics

There exists many ways to characterize what exactly a reinforcement learning algorithm is. In this subsection, three of the most basic characteristics which have the biggest implications on the performance

of an algorithm are identified and described, and are useful when explaining the observed behaviour of certain algorithms.

On-Policy vs Off-Policy

One basic distinguishing characteristic of reinforcement learning algorithms by being on-policy or off-policy. To be on-policy means to optimize for an estimated variable while simultaneously using the same variable to dictate the agent's actions, such a variable is often the policy function of an agent, hence the terminology, but can also be the value function of an agent.

In the case of the variable being a policy, an off-policy algorithm would be interacting with the environment using one policy and consequently generate transition tuples which are used as samples for training a second policy. The first policy which generates the transition tuples is called the *behaviour policy*, while the policy using the generated tuples as training samples is called the *target policy* [17].

Separating the sample generating and training policies can be desirable when considering the issue of exploration versus exploitation. When the two policies are separate, it is easier to direct the behaviour policy to be more exploratory, allowing transition tuples to cover a wider region of the state and action space and thus providing the target policy with greater generalization power. However, adopting an on-policy approach can result in a simpler algorithm, which converges to a stable policy much quicker albeit with a slightly less optimal agent [18].

Model-Based vs Model-Free

The second characteristic is model-based versus model-free, which is whether an algorithm uses some model of the environment's dynamics or not. For standard benchmarking of algorithms, it is possible to use the model of the benchmark environments, which are made with the intent of easy modelling, such as the inverted pendulum environment, the mountain car environment, or the lunar lander environment from the Farama foundation [19]. When such models are readily usable, an algorithm can leverage this model to efficiently learn the optimal value function and policy of the MDP.

However, in the real world, obtaining models of systems can be complicated. This is true in the aerospace industry where time and cost-intensive system identification campaigns are required to obtain a mathematical model describing the dynamics of a vehicle. To add further, novel aircraft designs which do not follow the traditional tube-and-wing design scheme, will not be able to directly leverage the wealth of developed tools or knowledge on flight dynamics for creating dynamic models. This traditional aircraft design scheme is very well known in the aviation industry, to the point where many well-established textbooks on the design of such aircraft and their flight dynamic properties are available, for instance: Torenbeek [20] on the synthesis of subsonic aircraft design, or Nelson [21] on flight stability and automatic control, amongst others. Such knowledge can, to some extent, be applied to the modelling and research of innovative aircraft designs, such as blended wing-body designs or distributed propulsion designs. However, the development of high-fidelity models for such aircraft designs is in itself a research area, where model developments have to stray away from familiar territories of tube-and-wing modelling, and is still being actively studied [22], [23].

In short, developing flight dynamic models for aircraft can be a drawn-out procedure, which complicates obtaining MDP environment models for flight control problems. When models of the environment are not at hand, a reinforcement learning algorithm can be designed in the following two ways:

1. Use some function approximator or pre-defined model structure and estimate the environment dynamics by updating this model, adding complexity in what model structure to use and steps for estimating the model.
2. To be model-free and only optimize value and policy functions based solely on sampled experiences, which is less efficient at learning than a model-based approach.

Value-Based vs Policy-Based

The distinction between value and policy-based is what the algorithm is designed to estimate. When the algorithm is value-based, it trains an estimate of the optimal value function from which the actions are inferred; versus in the policy-based case, where an estimate of the optimal policy function is trained from which actions are directly obtained.

In the case when function approximators are used in an algorithm, they are additionally distinguished by where the function approximator is used. Where value-based and policy based algorithms use the approximator in the value function and the policy function respectively.

The main benefits and drawbacks of each approach are in how they handle high dimensional states or action spaces, where value-based methods can handle a large number of states better and policy-based methods can handle a large number of actions better.

2.1.7. Basic Reinforcement Learning Algorithms

The goal of reinforcement learning algorithms is to teach an agent to behave optimally in some MDP. This is done by incrementally improving the value function estimate of a process, as well as the policy used to guide the agent. Three basic approaches can be identified to achieve this goal, these are *Monte Carlo Learning*, *Temporal Difference Learning*, and *Dynamic Programming*. The first two of these approaches are described in this subsection, while the last of these approaches is elaborated in more depth under Section 2.2.

In general, updating a value estimate involves using *targets*, which is the value that an estimate is moved towards, the most important distinction between Monte Carlo (MC) and Temporal Difference (TD) learning is in how this target is formulated.

Monte Carlo Learning

MC learning uses the *sampled episodes* from an agent interacting with an environment to improve its estimation of the value function. For illustrative purposes, the MC algorithm described here learns the action-value $Q(s_t, a_t) = Q_\pi(s_t, a_t)$ of the process.

Here, an agent initialized with a complete but random table of action-values along with an arbitrary policy is placed in a MDP and begins executing actions. With each action it takes, the environment transitions to a different state and the agent observes this new state along with the new reward. To allow the agent to learn, the action-value table is improved by summing up the rewards observed from each state-action pair that the agent comes across until this trajectory reaches a terminal state, and using this sum as a *monte carlo* estimate of the return G for the state action pair at the root of that trajectory.

$$Q(s_t, s_t) = Q(s_t, a_t) + \alpha[G_t - Q(s_t, a_t)] \tag{2.12}$$

Where α is a learning step size. This type of learning does not depend on the Markov property for convergence and the use of sampled experience as opposed to using modeled dynamics to estimate values makes MC learning appealing methods.

Temporal Difference Learning

TD learning also uses *sampled transitions* to improve an agent's action and value estimates. Unlike MC learning, it does not use sample episodes, thus it does not need to wait for the sum of rewards to reach a terminal state for a return's estimate to be defined, it updates the action-value of a state-action pair the instant which the agent receives the reward for that pair:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, a_{s+1}) - Q(s_t, a_t)] \tag{2.13}$$

Updating value estimates at every transition is known as *bootstrapping*, which is to improve the accuracy of an estimate by basing it on other estimates. Bootstrapping and temporal difference are widely applied methods in reinforcement learning because of their simplicity, and their minimal computation needs [17]. As opposed to MC methods, their wall clock time -real time needed- for training are often lower since they do not need to wait until the end of an episode for estimates to be updated.

2.2. Dynamic Programming

One major field of reinforcement learning research focuses on the use of *Dynamic Programming* techniques with an emphasis on tackling optimal control problems[24]. Dynamic programming is a term coined by Richard Bellman, it has roots in the field of optimization and is used to refer to the

problem-solving approach of divide and conquer, where a more complicated problem is broken into sub-problems for which recursive algorithms can be devised to come up with their solutions[16]. In the context of reinforcement learning, classical applications of dynamic programming revolve around the Bellman equation stated in Equation 2.10 and results in the method known as *Policy Iteration*, which is used to obtain the optimal value function and policy for *finite* MDPes, processes whose state s and action a variables belong to countable sets denoted $\mathcal{S}$ and $\mathcal{A}$ respectively. This method will be described in Section 2.2.1. Modern applications of dynamic programming extend the method of policy iteration to make it more computationally tractable and is described in Section 2.2.3.

The classes of algorithms that are branched off from dynamic programming methods are summarized in Figure 2.2.

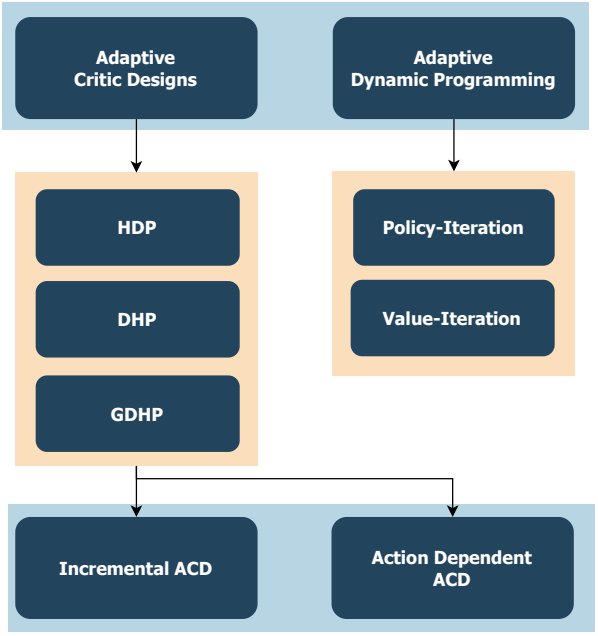


Figure 2.2: Overview of Approximate Dynamic Programming and Adaptive Critic Design algorithms, light blue box presents the class of algorithms, light orange box presents reinforcement learning algorithms.

2.2.1. Policy Iteration

This method iteratively applies two steps to find the optimal value function and policy, these steps are *policy evaluation* and *policy improvement*. The method is graphically depicted in Figure 2.3 and is outlined as follows:

1. Initialize $\pi(a|s)$ and $V_{\pi'}(s)$ arbitrarily.
2. Perform Policy Evaluation to compute the value function of π
3. Perform Policy Improvement to find a more optimal policy
4. Repeat from Step 2 until π and $V_{\pi}(s)$ have converged.

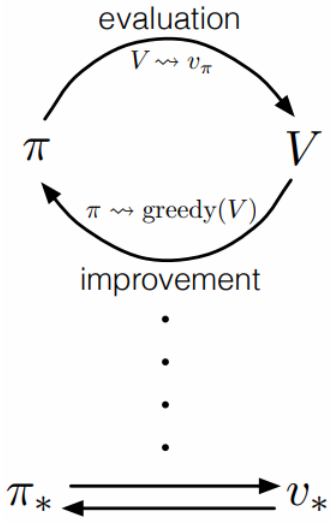


Figure 2.3: The method of policy iteration, by iteratively evaluation a policy's value function, and subsequently determining the greedy policy for that value function, the policy and value function eventually converges to a fixed iteration point when they are optimal for the given MDP[17]

This algorithm is proven to converge toward the optimal value function and policy[17] under certain conditions. Namely, it assumes a perfect environment model being available for use. Nonetheless, the theoretical guarantee for optimality makes it an important foundation method in reinforcement learning and is detailed further.

Policy Evaluation

For a given policy π, the value function of this policy V_π is computed by starting with an initial estimate $V_{\pi,0}$ and recursively applying the Bellman equation on all s as an update rule as shown in Equation 2.14. This update will be applied until the value function has converged, where convergence can be defined in several ways, the simplest of which is when the norm of the difference between subsequent value functions is smaller than a small constant ϵ.

$$V_{\pi,k+1} = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_{\pi,k}(s')] \quad \forall s \in \mathcal{S} \tag{2.14}$$

Note that to compute the update formulated in Equation 2.14, the environment dynamics $p(s', r|s, a)$ needs to be known and defined, which is an assumption made during the policy evaluation step.

By inspection of Equation 2.10, it can be seen that the update equation Equation 2.14 will only converge when the estimation v_k is equal to the true value function v_π.

Policy Improvement

For any given value function v_π, an improved policy π' is obtained by making π' deterministic and greedy with respect to v_π. This is done by defining π' to choose the action that has the highest action value in each state s:

$$\pi'(a|s) = \begin{cases} 1, & \text{if } a = \underset{a}{\text{argmax}}\ q_\pi(s, a) \\ 0, & \text{otherwise} \end{cases} \tag{2.15}$$

The policy improvement step is guaranteed to find a policy that is at least as good as the previous policy, and in the case of a finite MDP, the policy improvement step is guaranteed to find the optimal policy [16]. After finding a greedy policy on the given value function, the previous value function will no longer be accurate for the new policy *if it is suboptimal for the given MDP*. This means that when a policy evaluation step is subsequently performed, a sub-optimal policy will cause the evaluation step

to yield a different value function, but an optimal policy will yield the same value function. In other words, this algorithm only converges when the optimal policy and its corresponding value function are determined[17].

2.2.2. Generalized Policy Iteration

In the policy iteration method just described, the policy evaluation and improvement steps are carried out with the specific update rules in Equation 2.14 and Equation 2.15. However, the idea behind policy iteration: evaluating and improving upon some estimated policy or value function, is very useful and can be used to describe on a high level the idea behind many reinforcement learning algorithms as stated by Sutton and Barto [17], who use the term *generalized policy iteration* to capture this idea.

For example, instead of iterating the policy evaluation step until the value function converges, it is possible to only take one or a few policy evaluation iterations, and proceed with policy improvement thereafter. This is the idea behind Value Iteration algorithms, which is applied to optimal control problems without using Bellman equations[25], [26].

Another example comes from outside of the dynamic programming field, the class of actor-critic can be described using the idea of generalized policy iteration, wherein an estimate of the critic function (the value function) is improved based on sampled transitions from the MDP, and the policy function (the actor) is improved using information from the critic.

2.2.3. Approximate Dynamic Programming

Finite MDP are characterized by their discrete and small number of states and actions that are possible, this makes it possible to use dynamic programming to solve for the optimal value function and policy of such a process. However, many systems that are interesting to model and find solutions for have many states and actions possible, and real-life processes frequently have continuous variables instead of discrete ones. Such circumstances can cause a combinatorial explosion in the number of state-action pairs describing the system, which means having a value and an action probability distribution defined for each state becomes impractical. This is the so-called *curse of dimensionality*, alluded to in Section 2.2.1. Moreover, evaluation of the Bellman equations also requires a known transition probability distribution of the environment. Not only does this suffer from the curse of dimensionality in the case of large and/or continuous state spaces, but knowledge of such distribution might not be available for some systems, or at the very least inconvenient or difficult to identify.

One solution to circumvent these issues is to use function approximators for the value function, policy function, and environment model, which leads to the class of dynamic programming based reinforcement learning algorithms called Approximate Dynamic Programming (ADP). In this class, it is common to refer to value functions as *cost-to-go* or *cost* functions instead. Two basic methods exist in this class, the first is the Policy Iteration (PI) algorithm whose high-level algorithm is outlined in Algorithm 1.

Algorithm 1 Policy iteration, adapted from [27]

1: **Initialize**. Define a policy $\pi_0(s_k)$ which is admissible, i.e. stabilizing, and an arbitrary initial value function $V_0(s)$.

2: **Policy Evaluation**. Determine the value function of the current policy using the Hamilton-Jacobi-Bellman Equation:

$$V_{i+1}(s_t) = r(s_t, \pi_i(s_t)) + \gamma V_{i+1}(s_{t+1}) \tag{2.16}$$

3: **Policy Improvement**. Determine an improved policy.

$$\pi_{i+1}(s_t) = \underset{\pi(s_t)}{\operatorname{argmin}}[r(s_t, \pi_i(s_t)) + \gamma V_i(s_{t+1})] \tag{2.17}$$

4: If $V_i(s) \equiv V_{i-1}(s)$ then terminate, else $i = i + 1$ and return to step 2

Algorithm 1 fits into the mould of generalized policy iteration, which incorporates certain aspects of optimal control theory and has some note-worthy aspects. Firstly, the policy evaluation step Equation 2.14 of this algorithm is not formulated in the MDP framework like in Equation 2.14, and instead is the

Hamilton-Jacobi-Bellman (HJB) equation which is generally difficult to evaluate especially online[28] but can be solved approximately by methods such as least squares approximation using environment observations[29], or by iteratively evaluating the equation until convergence[27]. Secondly, instead of the policy being a probability distribution, this algorithm generally uses deterministic policies. Thirdly, if the system dynamics can be formulated as Equation 2.18 with n and m being the number of state and control variables respectively, and the reward of the system is formulated in a quadratic manner as shown in Equation 2.19, then it is known that the policy improvement step takes the form of Equation 2.20[30].

$$x_{t+1} = f(x_t) + g(x_t)u_t \tag{2.18}$$

$$r(x_t, u_t) = x_t^\top Q x_t + u_t^\top R u_t \tag{2.19}$$

$$\text{Where} \quad \{x, f(x)\} \in \mathbb{R}^n,\ g(x) \in \mathbb{R}^{n\times m},\ u \in \mathbb{R}^m,\ Q \in \mathbb{R}^{n\times n},\ R \in \mathbb{R}^{m\times m}$$

$$\pi_{i+1}(s_t) = -\frac{\gamma}{2} R^{-1} g^\top(s_t) \frac{\partial V_i(s_{t+1})}{\partial s_{t+1}} \tag{2.20}$$

The second basic algorithm of the ADP class is Value Iteration (VI), whose high-level algorithm is shown in Algorithm 2.

Algorithm 2 Value iteration, adapted from [27]

1: **Initialize**. Define an arbitrary policy $\pi_0(s_k)$, and an arbitrary initial value function $V_0(s)$.

2: **Policy Evaluation**. Determine the value function of the current policy using the Hamilton-Jacobi-Bellman Equation:

$$V_{i+1}(s_t) = r(s_t, \pi_i(s_t)) + \gamma V_i(s_{t+1}) \tag{2.21}$$

3: **Policy Improvement**. Determine an improved policy.

$$\pi_{i+1}(s_t) = \underset{\pi(s_t)}{\operatorname{argmin}}[r(s_t, \pi_i(s_t)) + \gamma V_i(s_{t+1})] \tag{2.22}$$

4: If $V_i(s) \equiv V_{i-1}(s)$ then terminate, else $i = i + 1$ and return to step 2

Here the same method of policy evaluation and reduction of policy improvement in the Linear Quadratic Regulator (LQR) case holds. The difference between PI in Algorithm 1 and VI in Algorithm 2 is that the policy evaluation step in Algorithm 2 involves only one evaluation of Equation 2.21, this is distinguished by the LHS value function having the subscript $i + 1$ and the RHS having i. Furthermore, VI do not require the initial policy to be admissible, which means that to use VI algorithms it is not necessary to perform any a priori step on the control policy.

Various forms of these algorithms have been developed further and shown a convergence rate that could allow them to be applied to online control of systems. For instance, Wei et al. [31] present an optimal control algorithm based on the framework of ADP using neural networks as function approximations. They demonstrate via simulation of an inverted pendulum control problem that even though their solution of the HJB equation is only approximately optimal, by virtue of iteratively performing the police evaluation and improvement steps, their ADP method can converge to the optimal control law within 5 s or 10 iteration steps. A similar method based on action-value or Q function is proposed by Lin et al. [32] and demonstrates similar convergence speeds.

By linearizing a nonlinear system's dynamics, it is possible to perform the policy evaluation and improvement steps in a much more efficient manner. This approach is adopted by Zhou et al. [33], [34] who used Recursive Least Squares (RLS) regression to identify a linear model at each time step, thus obtaining a time-varying linear model, which reduced the control problem to an LQR like problem. Note that this approach relies on a sufficiently high sensor sampling rate, in the order of 10^2 Hz, which is needed for the assumption of linear dynamics to hold, as any smooth nonlinear function can be approximated locally by linear functions.

Multi-step and Eligibility Trace Extensions of ADP Methods

One trade-off that ADP methods have to face is between adopting a more policy iteration or a more value iteration approach. In the former approach, algorithms developed generally see faster convergence in their estimates, naturally a desirable characteristic. However, such algorithms require the initial policies to be admissible, otherwise, the control policy would drive the system towards instability, causing the iterative HJB equation to grow unbounded. This issue is not faced by the latter approach, as VI algorithms do not seek to fully converge towards the optimal solution of the HJB equation, instead VI only require the value function estimation to be iterated by one step, instead of a large or infinite number of steps.

This dichotomy can be reframed into a spectrum of algorithms by considering the idea of *multi-step* iterations or updates, which is to take multiple transitions or timesteps worth of information while updating estimates in a reinforcement learning algorithm [17].

With the incorporation of a multi-step augmentation to the policy evaluation update rule, this dichotomy is turned into a spectrum of algorithms, where an ADP method can select how many steps should a policy evaluation iteration take. At the extreme of taking infinite steps, a multi-step ADP method is simply the PI algorithm, on the other extreme, a single-step ADP method is the VI algorithm [35].

A high-level view of the multi-step ADP algorithm is shown in Algorithm 3, with n being the number of steps. To automate the choice of n, Wang et al. [36] derived a criterion for switching n from a low number during the initial iterations of the algorithm, where the initial arbitrary policy is not guaranteed to be admissible, to a high number during the latter iterations when the policy is verified to be admissible under the proposed criterion.

Algorithm 3 Multi-step value iteration, known as multi-step heuristic dynamic programming in and adapted from [35].

1: **Initialize**. Define an arbitrary policy $\pi_0(s_k)$, and an arbitrary initial value function $V_0(s)$.
2: **Policy Evaluation**. Determine the value function of the current policy using the multi-step approximation of the Hamilton-Jacobi-Bellman Equation:

$$V_{i+1}(s_t) = \gamma^n V_i(s_{t+1}) + \sum_{l=t}^{t+n-1} \gamma^{l-t} r(s_l, \pi_i(s_l)) \tag{2.23}$$

3: **Policy Improvement**. Determine an improved policy.

$$\pi_{i+1}(s_t) = \underset{\pi(s_t)}{\operatorname{argmin}}[r(s_t, \pi_i(s_t)) + \gamma V_i(s_{t+1})] \tag{2.24}$$

4: If $V_i(s) \equiv V_{i-1}(s)$ then terminate, else $i = i + 1$ and return to step 2

In a similar vein of leveraging the loose policy condition which Heuristic Dynamic Programming (HDP) has, and finding ways of increasing the convergence rate of HDP, eligibility traces can also be employed [37], [38]. Eligibility traces are an alternative method of incorporating additional samples from past timesteps for updating estimates, the objective here is again to improve convergence rates of algorithms, however, the algorithm behind this method is different than multi-step ideas.

Instead of explicitly retrieving samples from certain time steps, eligibility traces simply store the past updates made to the critic or actor, and persistently but at a decaying rate add such updates to the critic or actor for subsequent timesteps. This method is illustrated in Equation 2.25, where $\theta(t)$ is the set or vector of parameters for a certain function approximator at time t, this approximator could be the critic or the actor, $\mathcal{E}(t)$ is the eligibility trace at time t, and $\Delta\theta(t)$ is the update made to the function approximator at time t.

$$\begin{aligned} \theta(t+1) &= \theta(t) + \mathcal{E}(t) \\ \mathcal{E}(t+1) &= \mathcal{E}(t) + \nabla\boldsymbol{\theta}(t), \quad \mathcal{E}(0) = 0 \end{aligned} \tag{2.25}$$

2.2.4. Adaptive Critic Designs

Alongside adaptive dynamic programming, there is a class of algorithms called Actor-Critic Design (ACD). In this class of algorithms, the estimate of value functions is referred to as the *critic*, the critic is thus "responsible" for policy evaluation; while the estimate of the policy functions is referred to as the *actor*, which is thus "responsible" for policy improvement. It should be noted that in literature, the terms ADP and ACD can be found to be used interchangeably [39]–[41], notably with the progenitor of ACD Werbos also using these terms and reinforcement learning interchangeably [25]. Algorithmically, practical implementations of some ADP and ACD algorithms are very much alike, as will be stated later during the description of HDP. However, in this literature study, the distinction between ADP being dynamic programming algorithms which are more optimal control-oriented, and ACD being dynamic programming algorithms which are more reinforcement learning-oriented is made.

Adaptive critic designs can be considered to be reinforcement learning algorithms developed from generalized dynamic programming algorithms [41] whose algorithms are structured similarly to the PI Algorithm 1 or the VI Algorithm 2. Just like ADP, ACD algorithms are sample efficient enough for entirely online trained controllers to be implemented, which can converge towards a stable controller within a short period of time [35], [39], [42].

Several main algorithms form the basis of this class: the first and simplest algorithm is HDP, the second and slightly more complicated algorithm is Dual Heuristic Programming (DHP), and the third but most complicated algorithm is Globalized Dual Heuristic Programming (GDHP). The increase in complexity comes from what the critics of each algorithm estimate, where HDP only estimates the value function, DHP estimates the gradient of the value function, and GDHP estimates both the value and gradient of the value functions. Two extensions of all three of these basic algorithms exist, the first kind of extension makes the critic function Action-Dependent (AD), which changes the critic from being an estimate of the state-value function to an estimate of the action-value function [43]. The second kind of extension changes how the model estimate is obtained, where an RLS regressor is used to identify linear systems for the immediate time steps, as opposed to using online supervised learning with neural networks or with offline identified models.

In the ACD context, the value function is often called the cost-to-go function, but its definition remains the same as the return expected from a given state; the rewards also have a different name, and are sometimes called the one-step costs.

$$V(s_t) = r_t + \gamma V(s_{t+1}) \tag{2.26}$$

Heuristic Dynamic Programming

HDP is characterized by using the critic to estimate the value function $V(s_t)$ itself:

This critic is evaluated by the critic TD error $e_c(t)$ defined in Equation 2.27, and is trained to minimize the critic error function Equation 2.28. To perform training, the function approximator of the critic is updated using a gradient descent method to minimize the error function, meaning the parameters $\theta(t)$ are updated according to Equation 2.29.

$$e_{1,c}(t) = V(s_t) - r_t - \gamma V(s_{t+1}) \tag{2.27}$$

$$E_c(t) = \frac{1}{2} e_{1,c}(t)^\top e_{1,c}(t) \tag{2.28}$$

$$\theta_c(t+1) = \theta_c(t) - \nabla\theta_c(t) \tag{2.29}$$

$$\text{Where} \quad \nabla\theta_c(t) = \eta_c \frac{\partial E_c(T)}{\partial \theta_c(t)} = \eta_c \frac{\partial E_c(T)}{\partial e_{1,c}(t)} \frac{\partial e_{1,c}(t)}{\partial V(s_t)} \frac{\partial V(s_t)}{\partial \theta_c(t)}$$

Correspondingly, there is also the actor loss, the actor error function, and the actor update rule that are expressed in the following equations. Note that the notation of s and x for MDP and model state is used interchangeably on the assumption that the two state variables are equivalent.

$$e_a(t) = V(s_t) - V_{true}(s_t) \tag{2.30}$$

$$E_a(t) = \frac{1}{2} e_a(t)^\top e_a(t) \tag{2.31}$$

$$\theta_a(t+1) = \theta_a(t) - \nabla\theta_a(t) \tag{2.32}$$

$$\text{Where} \quad \nabla\theta_a(t) = \frac{\partial E_a(t+1)}{\partial \theta_a(t)} = \frac{\partial E_a(t+1)}{\partial e_a(t+1)} \frac{\partial e_a(t+1)}{\partial V(s_{t+1})} \frac{\partial V(s_{t+1})}{\partial s_{t+1}} \frac{\partial x_{t+1}}{\partial u_t} \frac{\partial u_t}{\partial \theta_a(t)} \tag{2.33}$$

Observing the formulation of HDP thus far presented, an interesting note can be made of the close resemblance in the practical implementation of HDP and PI or VI algorithms or even the interchangeable use of ADP with HDP [35], [44], [45].

Observing the update equations, it can be seen that the actor update contains the partial derivative $\frac{\partial x_{t+1}}{\partial u_t}$, this is a derivative that needs to be obtained using a system model which would define the relation between the state x_{t+1} and the action u_t. As a result, this makes the HDP algorithm model-dependent. However, this dependency can be removed if HDP is made AD, in which case the value function would be a function of both state and action $V(s_t, u_t)$, this allows the chain rule expansion in Equation 2.33 to reduce the term $\frac{\partial V(s_{t+1})}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial u_t}$ to only $\frac{\partial V(s_{t+1})}{\partial u_t}$ since V would be an explicit function of u_t, thus eliminating the system dynamics $\frac{\partial x_{t+1}}{\partial u_t}$.

An alternative way of alleviating model dependency is to use the incremental method of Zhou et al., who developed the Incremental Heuristic Dual Programming (IHDP) algorithm and successfully applied it to several tasks, including a flight control task [46], and a launch vehicle control task [47]. In this case, the partial $\frac{\partial x_{t+1}}{\partial u_t}$ can be replaced by a control effectiveness matrix from an online identified linear system. Note that this does not make the algorithm entirely model free, since the formulation of the update rules still involves using system dynamics, i.e requires environment modeling.

This algorithm performs relatively poorly compared to DHP and GDHP in terms of control performance and disturbance rejection [41], [48], but nonetheless have been deployed successfully [49]–[51].

Dual Heuristic Programming

DHP is characterized by the critic estimating the gradient of the value function $\lambda(s_t)$, instead of the value function directly. This gradient can also be referred to as the *costate* [27]:

$$\lambda(s_t) = \frac{\partial V(s_t)}{\partial s_t} \tag{2.34}$$

Here, the actor formulation is identical to the HDP algorithm, and only the critic formulations are changed. The critic TD error is now defined using gradients of the value function, as shown in Equation 2.35, with the critic error function and function parameter update remaining unchanged.

$$e_{2,c}(t) = \lambda(s_t) - \frac{\partial r_t}{\partial x_t} - \gamma\lambda(s_{t+1}) \frac{\partial x_{t+1}}{\partial x_t} \tag{2.35}$$

$$E_c(t) = \frac{1}{2} e_{2,c}(t)^\top e_{2,c}(t) \tag{2.36}$$

$$\theta_c(t+1) = \theta_c(t) - \nabla\theta_c(t) \tag{2.37}$$

$$\text{Where} \quad \nabla\theta_c(t) = \eta_c \frac{\partial E_c(t)}{\partial \theta_c(t)} = \eta_c \frac{\partial E_c(t)}{\partial e_{2,c}(t)} \frac{\partial e_{2,c}(t)}{\partial \lambda(s_t)} \frac{\partial \lambda(s_t)}{\partial w_c(t)}$$

With this variation, the model-dependency of the algorithm has increased, as the critic TD error also uses the system dynamics in the form of $\frac{\partial x_{t+1}}{\partial x_t}$ in its formulation. Here, introducing an AD variant will not remove the model dependence from the algorithm entirely, only from the actor component.

Because the critic function in DHP directly outputs value function gradients, which are needed in actor updates, there are no additional numerical errors that get injected into the actor function parameter update, which cannot be said for the HDP algorithm.

This algorithm is extended to an incremental model identification version, resulting in IDHP. Application of IDHP to the task of flight control demonstrated improved reference tracking performance and fault tolerance than a pure DHP algorithm [4], [52], wherein the IDHP controller was able to recover control of the simulated aircraft even after the flight dynamics were reversed mid-flight.

Globalized Dual Heuristic Programming

GDHP is characterized by the critic estimating the value and gradient of the value function simultaneously, thus the critic can be treated as returning a vector of value function variables $\begin{bmatrix} V(s_t) & \lambda(s_t) \end{bmatrix}^\top$.
This results in the GDHP critic error function being composed of two terms, a HDP and a DHP td error

$$E_c(t) = \frac{1}{2} e_{1,c}(t)^\top e_{1,c}(t) + \frac{1}{2} e_{2,c}(t)^\top e_{2,c}(t) \tag{2.38}$$

$$\theta_c(t+1) = \theta_c(t) - \nabla\theta_c(t) \tag{2.39}$$

$$\text{Where} \quad \nabla\theta_c(t) = \eta_c \frac{\partial E_c(t)}{\partial \theta_c(t)} = \eta_c \left(\frac{\partial E_c(t)}{\partial e_{1,c}(t)} \frac{\partial e_{1,c}(t)}{\partial V(s_t)} \frac{\partial V(s_t)}{\partial \theta_c(t)} + \frac{\partial E_c(t)}{\partial e_{2,c}(t)} \frac{\partial e_{2,c}(t)}{\partial \lambda(x_t)} \frac{\partial \lambda(s_t)}{\partial \theta_c(t)} \right) \tag{2.40}$$

GDHP are theoretically superior to both HDP and DHP since its critic estimates both the outputs of their respective critics, but the two common implementations of GDHP both have practical issues. In the first common implementation, the GDHP critic only outputs the value function [53], [54] and the partial derivative $\frac{\partial \lambda(s_t)}{\partial w_c(t)}$ from Equation 2.40 is then computed by replacing it with the partial derivative $\frac{\partial^2 V(s_t)}{\partial s_t \partial w_c(t)}$, which is computationally expensive and practically complicated to implement. The second common implementation of GDHP involves using one function approximator for the critic, which outputs both the value function and its gradient [55]–[57], just as stated at the beginning of this subsection. However, in such implementations, the gradient of the value function is not guaranteed to be an accurate estimate of the gradient.

Reconciling the two issues of computational complexity and analytical accuracy, Zhou [58] proposed the idea of building the critic as two function approximations but with a novelty of formulating the $\lambda(s_t)$ approximator based explicitly on the $V(s_t)$ approximator, which was applied to a longitudinal flight control task, and shows promising capacity in being able to reap the simultaneous benefit of efficiency and accuracy.

2.3. Deep Reinforcement Learning

Besides Adaptive Dynamic Programming, the other major field of reinforcement learning research is Deep Reinforcement Learning (DRL). Both of these fields revolve around using function approximators [59] to address the curse of dimensionality which hinders the deployment of reinforcement learning algorithms to real-life problems, which often contain many continuous variables. A big distinction between these fields is how DRL stems largely from deep learning methods, where deep neural networks are used as the function approximator or incorporated in any other way into algorithms. The types of algorithms belonging to this class are summarized in Figure 2.4. As can be seen, DRL is separated into various classes of algorithms and this section will describe each one of these classes, in addition to presenting notable algorithms from each class.

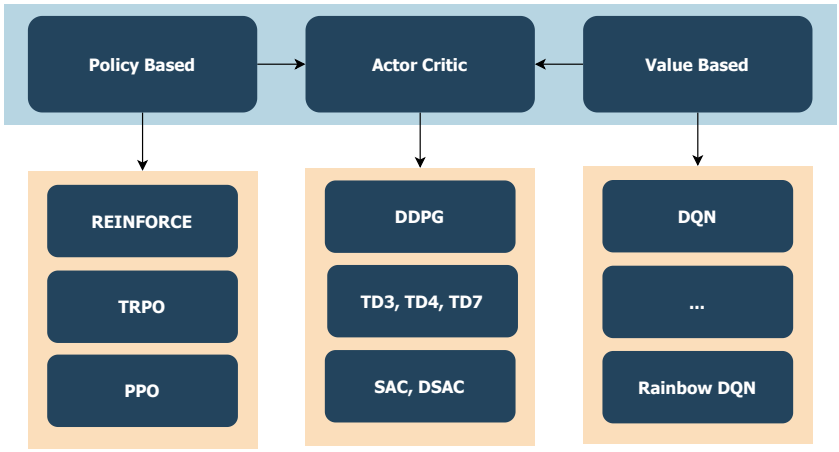


Figure 2.4: Overview of deep reinforcement learning algorithms, light blue box presents the class of algorithms, light orange box presents reinforcement learning algorithms.

This section is laid out as follows, first a description of deep learning is given in Section 2.3.1, then value-based methods are discussed in Section 2.3.2, followed by policy-based methods in Section 2.3.3, and lastly a discussion on the combination of policy and value-based methods called actor-critic methods in Section 2.3.4.

2.3.1. Deep Learning

Deep learning refers to the subfield of machine learning which studies deep neural networks and their applications. This field has its origins in the ideas of artificial neural networks and is a scientific effort that has dramatically changed the idea of what an artificial neural network is and what it might be capable of. Deep neural networks fundamentally are an expansion of artificial neural networks, they use the same basic architecture of a neural network, with layers of interconnected neurons between a surface or input layer and a final output layer. One distinction between deep and artificial neural networks is that deep neural networks use many more layers and nodes than are typical in artificial networks. In addition to deep neural networks, deep learning also encompasses many other types of neural networks, a short list of some network types under deep learning and their descriptions are listed below.

1. **Deep Neural Network (DNN)**. Neural networks with a large number of neurons and hidden layers. From the *Universal Approximation Theorem*, it is known that neural networks are capable function approximators, and were pioneered to do as such [60]. The large network structure of DNNs allows them to be trained more efficiently to approximate a given function [61], in addition to allowing them to approximate more nonlinearities.
2. **Convolutional Neural Network (CNN)**. Neural networks that contain convolutional layers that processes the input into a map of features detected. Pioneered in handwritten number detection [62], incorporating convolution into neural networks allows for detecting and utilizing patterns that may exist in the input data.
3. **Recurrent Neural Network (RNN)**. Neural networks that feed their output back as input, or have some form of memory component. Their architecture makes them more applicable to temporal data than other network architectures and is popular for generative predictive sequences in for example text writing [63] and language modeling [64].

Deep learning had several groundbreaking results before practitioners of reinforcement learning managed to apply some of the advances into their own algorithms, such as in the area of computer vision [65], [66], speech recognition [67], [68], and in handwriting generation [63]. These results demonstrated that deep neural networks had the power to use the same raw data which humans perceive, and interpret useful information out of them, or even generate new information. So it seemed only natural to see if it was possible to combine the learning emulation of reinforcement learning, with the world processing emulation of deep learning, to produce machines that could perceive *and* learn from the world just as humans do. This ethos is notably different from that of ADP or ACD, as here the emphasis is on

recreating the way which humans and animals interact with an environment, rather than on extending mathematical theory.

DNNs for Function Approximation

A generic neural network layer can be denoted as a vector function $f(x)$ in the following manner:

$$f(x) = g(\theta x + b) \tag{2.41}$$

With $\boldsymbol{\theta}$ a square weight matrix, b a vector of biases, and $g(\cdot)$ an arbitrary activation function, preferably differentiable. To build a full network such as a DNN, one simply has to use the output of one layer as the input of a subsequent layer:

$$y = h(x) = f_1(f_2(\ldots(f_k(x)))) \tag{2.42}$$

Just as the case in ADP and ACD, when DNN is being used as the function approximator, the weights and biases parameterizing the network are the values that need to be updated during training. These updates are done in such a way that improvements in estimate accuracy are observed, which can be done by gradient descent.

The specific gradient descent method used to optimize parameters in neural networks is called *back-propagation*, which is the name given to the broad method of determining the derivative of the entire set of parameters. This method uses, the gradient of the network's *loss function* J, which measures the accuracy of the network's prediction, with respect to all the weight and biases of the network, to provide the increment on each parameter that would result in the lowest loss. When $g_k(\cdot)$ is differentiable everywhere, this gradient will always be defined, when that is not the case, numerical errors may propagate through the network update thus differentiability for $g(\cdot)$ is generally desired.

This partial derivative in practice can be calculated very efficiently, as all the information needed is present during one feedforward step of the network.

For the main step of optimizing values of weights and biases, this gradient is used with an optimizer to determine the increment applied to each parameter [69]. A popular option in DRL is Stochastic Gradient Descent (SGD), the stochastic in this optimizer's name refers to the fact that this optimizer uses one training sample to determine a gradient, which is stochastic as the training sample is essentially one realisation of a random process. This single sample gradient is then scaled by a learning rate η SGD is an easy-to-understand and implement optimizer, but is rudimentary in comparison to other optimizers. SGD nonetheless forms the basis for many gradient optimizer algorithms, and the parameter update rule using SGD is shown as follows:

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t) \tag{2.43}$$

One improved optimizer is the Adagrad optimizer, which is based on SGD but features a variable η which adapts to each parameter, such that the size of a parameter's update is based on the frequency that the parameter is updated; these changes make Adagrad particularly suitable when training data is sparse [69].

Finally, when it comes to using DNNs as function approximators, one should keep in mind that they break many of the proven convergence properties that reinforcement learning algorithms would otherwise possess with a linear function approximator, or no function approximation at all [17], [70]. In addition, the optimization problem of finding the optimal function approximator parameters become more difficult to solve when using DNNs. For example, the optimization problem is typically multi-modal which means that it is possible for optimization algorithms to be stuck at local optimum. This does not neccessarily mean it is infeasible to use DNNs for function approximation, as long as proper hyperparameter tuning and addition of convergence/stability aiding measures are used, as will be made evident in subsequent sections.

2.3.2. Value Based Deep Reinforcement Learning

Value-based DRL algorithms focus on the estimation of the optimal value function $Q^*(s, a)$ using DNNs as function approximators, resulting in an approximate value function $Q(s, a, \theta_c)$ parameterized by some

parameters θ_c, and have a relatively simple policy component that picks the action which has the maximum value according to the value estimates, i.e. picks the greedy action. It is sometimes desirable to not always act greedily, and to instead pick the actions estimated to be suboptimal with some small probability ϵ in order to facilitate exploration, a so-called ϵ greedy strategy:

$$\pi'(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \underset{a}{\text{argmax}}\, Q_\pi(s,a) \\ \epsilon, & \text{otherwise} \end{cases}, \quad 0 < \epsilon < 1 \tag{2.44}$$

Deep Q Network

The first successful DRL algorithm came from the value-based approach, and it was the Deep Q Network (DQN) algorithm [71]. It was a model-free algorithm that managed to successfully use pixel-based sensory input to learn control policies for a series of Atari games. The algorithm is based on Q-learning, in the tabular case this would mean using either the temporal difference learning method or the MC learning method to iteratively update an estimate for the action-value function of all state-action pairs. Here, the dimension of the state space was immense, because frames of gameplay footage are used as pixel-based sensory information. Thus it is logical to use a CNN as a function approximator for the action-value function, called the *Q-network*, instead of discretely recording the action value for every combination. This algorithm had two features that contributed largely to its success, they were: *Expereience Replay* and *Target Network*.

- **Experience Replay** [71]. Deep learning typically assumes samples to be independent, which is untrue for samples in reinforcement learning problems and can result in biases introduced to the Q-network estimates. To reduce the correlation between samples, the agent's experienced transition tuples (s_t, a_t, r_t, s_{t+1}) are stored in a queue memory-buffer with a limited size n; and training of the Q-network is done by sampling a transition randomly from this queue, and updating the value estimate associated with the state transition of this sample with the recorded reward.
- **Target Network** [71]. The target here is the value that a value estimate is moved towards, as described in Section 2.1.7. Learning instability can arise when targets change too rapidly, which can be the case if the targets are taken from the estimated Q-network. Hence, a separate *target* Q-network is stored, from which the update targets are retrieved and whose weights are only periodically synchronized with the estimated Q-network. Thus ensuring the targets do not vary drastically from update to update.

The problems that these two features aim to tackle are universal in reinforcement learning, and while other techniques to address these challenges of sample correlation and learning stability exist, the two features augmented to DQN can be useful in other algorithms as well.

Rainbow DQN and other improvements

Many independent experiments and improvements were made to the architecture of DQN and other reinforcement learning algorithms, each of which addressed an issue separate from the other, van Hasselt et al. compiled a list of such ingredients for an algorithm and combined them into one single DQN value based algorithm called Rainbow DQN [72]. These various modules are compiled in the following list:

- **Double Learning** [73]. Originally proposed in the tabular setting, the idea of double learning is to keep and learn two estimates of the value function at the same time, with either one of the two networks being randomly chosen to be trained for any given experience sample. Doing so reduces the value estimates, which are overly optimistic due to the bias arising from the usage of the argmax operator [74].
- **Prioritized Experience Replay** [75]. When learning under the experience replay framework, the transitions from the memory buffer are sampled with uniform probability. This can be suboptimal considering that some transitions do not contain significant information on the environment, and thus do not train the agent as well as it would have been trained if another transition was sampled. By assuming transitions that result in a higher TD error as being more important for learning, and

sampling such transitions with higher probability, it has been shown that an agent's rate of learning improves, i.e. sample efficiency can be raised.

- **Duelling Network Architecture** [76]. A novel neural network architecture, named *duelling architecture*, is adopted to approximate the value function estimates. The duelling architecture is distinguished by the output layer being preceded by two streams of separate hidden layers, the first stream being used to estimate state-value functions, while the second is used to estimate action advantages -a similar variable to the action value-. Preceding these two streams is one common convolutional network, laid out in a similar manner as the convolutional layers in the original DQN algorithm. By utilizing this duelling architecture, the complexity of learning for the neural network is reduced, which contributes towards higher sample efficiency and better agent performance.
- **Multi-step Bootstrapping** [17], [77]. The basic approach in bootstrap learning algorithms and thus in most value-based methods is to use the reward from a single transition as the learning target. As suggested by Sutton [78] and corroborated in ADP research [35], taking multiple timesteps worth of transitions to form a learning target can produce faster learning rate, as it allows rewards to be propagated throughout the domain of the estimate faster [78]. This variation was used on four standard algorithms by Mnih et al. [77], where the idea was brought further to using transitions from several asynchronously trained agents, which defined a class of asynchronous algorithms with agents trained on separate computing units.
- **Distributional Learning** [79], [80]. Whereas the traditional reinforcement learning paradigm is concerned with *expected* values, there is also a view that studying the distribution of values can be incorporated into learners, which should allow for more risk-aware behaviours, and intuitively allow agents to make use of more information than merely the expectation of a random variable. A commonly used analogy is that distributional learning is to treat the environment observations as colours pictures, while expectational learning is to treat it as black-and-white pictures. To make use of distributional reinforcement learning, an estimate for the distribution of returns and thus of the state or action-values are learned, specifically the first and second moments of the distribution [79].
- **Noisy networks** [81]. The problem of exploration vs exploitation is important for algorithms to address, noisey networks seek to provide a means of finetuning the degree of exploration built into an algorithm. A noisy network is made by adding random values to the weights and biases to any neural network, which can be done to any given network as well as other forms of function approximators. The advantage that such an augmentation brings to algorithms is by allowing estimates to escape local optima, increasing the likelihood of encountering the global optimum by virtue of traversing the domain more widely.

2.3.3. Policy Based Deep Reinforcement Learning

Policy-based methods, also known as policy search methods, primarily optimize the policy of an agent as opposed to the value function, and can select actions without the need to consult value functions, but can reap benefits by simultaneously estimating values as will be discussed in Section 2.3.4. The policy $\pi(a|s, \theta_a)$ in such methods is approximated with a DNN parameterized with the parameters θ_a. By training for an optimal policy directly, instead of training a value function on which a policy will be inferred, one layer of complexity is removed from the training task.

The foundation of policy-based methods is based upon the *policy gradient theorem*, which is an analytical expression derived from the Bellman equations of Equation 2.10 which defines the gradient of a policy's performance or optimality with respect to the policy parameters, this gradient is called the *policy gradient*. According to this theorem, the following proportionality for the gradient of a policy's loss holds true:

$$\nabla J(\theta_a) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \theta_a) \tag{2.45}$$

Where $\mu(s)$ is the state distribution which describes the importance of each state, which is related to how often a state is visited, and thus is theoretically a function of the policy parameters since the policy decides what action and thus what states might be visited. The Important result of this theorem is that this expression shows that policy gradient does not require taking the gradient of $\mu(s)$, despite it being

technically a function of policy parameters, which makes it much easier to implement gradient descent routines to optimize a policy [17].

REINFORCE

The simplest form of policy-based method is called REINFORCE, which is an acronym for the form of the algorithm: REward Increment = Nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility [82]. REINFORCE uses the MC learning approach to train its policy network, where the parameter updates to the policy are only performed at the end of each episode. The parameter update rule of REINFORCE is expressed as follows [17]:

$$\theta_{t+1} = \theta_t + \eta G_t \nabla \ln(\pi(A_t|S_t, \theta_t)) \tag{2.46}$$

In this update rule, every parameter update is proportional to the return observed G_t for each state-action pair $S_t A_t$., and it is in the direction of the vector represented by $\nabla \ln()$, which is the direction in parameter space that contains policies which have a higher likelihood of repeating action A_t when in state S_t. This update is thus logical since if a state action pair has a high return, it would be beneficial to update the policy to enact such actions more, which is exactly what the update term in Equation 2.46 does.

Trust Region-policy Optimization

The theoretical convergence properties of REINFORCE are very positive, in the expectation this algorithm is guaranteed to improve the policy [17], [82]. In practice, algorithms revolving around the REINFORCE framework have very low sample efficiency and thus are slow to converge, in addition to high variance in learning rate. One way of circumventing these deficiencies is to incorporate baselines that modulate the magnitude of policy parameter updates, thus reducing learning variance. An extension of this idea is the TRPO algorithm proposed by Schulman et al.

TRPO effectively reduces the variance of policy updates through constraining update steps when the parameter updates are too aggressive [83], this aggressiveness is determined by measuring the distance between the original policy and the updated policy. Since a policy function is a probability distribution, measuring the distance between policies thus requires using probability distance measures, in the case of TRPO this distance measure is the Kullback-Leibler (KL) divergence; returning a scalar value representing how different the two probability distributions are [84].

This algorithm casts the policy-gradient method as an optimization problem, which is formulated as follows:

$$\begin{aligned} &\underset{\theta_a}{\text{maximize}} \ \mathbb{E}\left[\frac{\pi(a|s,\theta_a)}{\pi(a|s,\theta_{a,old})} Q_{\pi_{old}}(s,a)\right] \\ &\text{subject to} \ \mathbb{E}\big[D_{KL}\big(\pi(\cdot|s,\theta_a)||\pi(\cdot|s,\theta_{a,old})\big)\big] \leq \delta \end{aligned} \tag{2.47}$$

Being a policy-based method, no function is used to approximate the action-values $Q_\pi(s,a)$. Instead, π is executed over some number of time steps generating a trajectory, and these values are estimated using samples from the trajectory.

Formulating the policy learning problem as an optimization problem is noticeably divergent from all algorithms thus far presented, indeed this approach of addressing the reinforcement learning problem is a field of research in itself, such as in the DRL subfield of policy search [85] and evolutionary reinforcement learning [86]. The optimization problem from Equation 2.47 is then solved using conjugate gradient optimization algorithms. The specific optimization algorithm proposed by Schulman et al to solve the TRPO problem uses Hessian matrices and thus can be considered to be a second-order algorithm.

Despite the reduced learning variance benefits that this algorithm has over REINFORCE, TRPO remains complicated to implement, uses the KL divergence measure which is computationally expensive due to the optimization algorithm posed, while suffering from slow learning rates [87]. Hence, more augmentations were proposed.

Proximal Policy Optimization

When the policy-gradient algorithms were cast into an optimization problem, as done by Schulman et al. through TRPO, this opened up new possibilities for making changes to policy-gradient algorithms. For one, instead of posing a constrained optimization problem as done in Equation 2.47, an unconstrained optimization problem can be formulated by stating the optimization constraints as penalties in the objective function being maximized, resulting in the optimization problem posed by Equation 2.48 with a tunable parameter β.

$$\underset{\theta_a}{\text{maximize}} \ \mathbb{E}\left[\frac{\pi(a|s,\theta_a)}{\pi(a|s,\theta_{a,old})} A_{\pi_{old}}(s,a) - \beta D_{KL}[\pi(\cdot|s,\theta_a)||\pi(\cdot|s,\theta_{a,old})]\right] \tag{2.48}$$

In fact, Equation 2.48 is the optimization problem suggested by the theory which justifies TRPO [87]. However, the presence of β makes this objective function difficult to define, thus surrogate objective functions are formulated to pose essentially the same optimization problem as Equation 2.48. Such a surrogate objective function $J_{CLIP}(\boldsymbol{\theta}_a)$ is defined in Equation 2.49, it has a tunable parameter ϵ and uses the *advantage function* $A_\pi(s,a)$, and is the loss function that creates the PPO algorithm.

$$\begin{aligned} \underset{\theta_a}{\text{maximize}} \ J_{CLIP}(\theta_a) =& \mathbb{E}\left[\min\Big(r(\theta_a), \text{clip}\big(r(\theta_a); 1-\epsilon, 1+\epsilon\big)\Big) A_{\pi_{old}}(s,a)\right] \\ \text{Where} \quad r(\theta_a) =& \frac{\pi(a|s,\theta_a)}{\pi(a|s,\theta_{a,old})}, \quad A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \end{aligned} \tag{2.49}$$

Just like TRPO, PPO uses samples from generated trajectories to estimate the advantage function values. By using $J_{CLIP}(\theta_a)$, first-order optimization techniques such as SGD can be adopted, which are much more computationally efficient than second-order techniques. It is with such an objective function that PPO can have the variance reduction of TRPO but with better computational complexity and empirically better sample efficiency as shown in Figure 2.5.

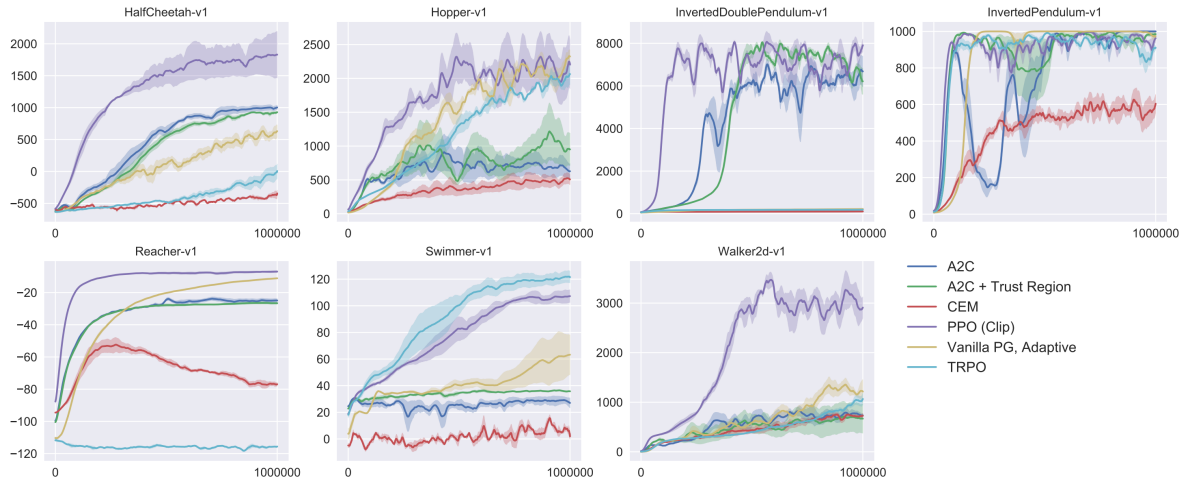


Figure 2.5: Comparison of PPO with TRPO and various other algorithms on several benchmark environments, taken from [87]

2.3.4. Actor Critic Deep Reinforcement Learning

With both value-based and policy-based methods introduced, the backdrop is set for actor-critic methods. In both these predecessor methods, either the value function or the policy is being approximated by a DNN, this has the big disadvantage of still being ham-strung by the curse of dimensionality when either action or state space is large. Specifically, value-based methods can perform well when the state space is large and the action space is small, while policy-based methods perform well when the action space is large and the state space is small.

For continuous control problems with continuous state and action spaces, both cases are true. The logical solution is, therefore, to use function approximators for both value and policy functions, with DNN being the popular approximator candidate. This architecture is called *actor-critic*, where the *actor* refers to the policy or the DNN approximating the policy, and *critic* refers to the value function or the approximating DNN. They can also commonly be referred to as actor/critic-networks.

Moreover, because actor-critic method models both value and policy functions, such methods typically involve improving one function after the other. This structure is reminiscent of the generalized policy iteration idea from dynamic programming, mentioned in Section 2.2.2, showing that interestingly many ideas within reinforcement learning build upon and are related to one another.

Deep Deterministic Policy Gradient

Lillicrap et al. identified many of the same problems stated in this subsection's introduction, and combined the experience replay and target network ideas of DQN for value function training, along with the ability of policy-based algorithm to handle continuous action spaces, to build the actor-critic algorithm known as DDPG [88].

The core algorithm of DDPG is based on Deterministic Policy Gradient (DPG), this algorithm is already an actor-critic algorithm but does not use deep neural networks for any function approximation, hence the lack of *deep* in DPG. DPG uses a deterministic policy as opposed to a stochastic policy, this results in more efficient gradient determinations as the gradient of deterministic policies only requires integrating over the state-space, rather than over the state-*action*-space which is required for gradients of stochastic policies [70]. An intuitive explanation behind this difference is that stochastic policies have a non-zero probability of taking each action in the action-space, thus to evaluate the change in optimality of such a policy, it is necessary to consider the entire action space, the same is not true for deterministic policies since they can only take one action for any given state.

DPG in it's original form is incompatible with using DNN for function approximation, as using such approximations results in the applied gradient computations becoming incorrect estimations of the true policy gradient [70]. However, by applying the tricks of experience replay and target networks from DQN, DDPG is able to extend DPG to employ DNN for function approximation as well. Specifically, Lilicrap et al. used replay buffers to train their actor and critic-networks, in addition to creating target networks for both the critic and actor thus stabilizing learning.

The crux of DDPG algorithm is in how it trains the actor and critic. This is done by sampling a minibatch of N transition tuples $(s_i, a_i, s_{i+1}, r_i)$from the replay buffer, and using this minibatch of samples to determine the loss function of Equation 2.50 which the critic is trained to minimize, and the policy gradient Equation 2.52 used to update the actor.

$$J_c = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i, \theta_c))^2 \tag{2.50}$$

$$y_i = r_i + \gamma Q(s_i, \pi(s_{i+1}, \theta_a^T), \theta_c^T) \tag{2.51}$$

$$\nabla_{\theta_a} J_a = \frac{1}{N}\sum_i \nabla_a Q\big(s_i, \pi(s_i, \theta_a), \theta_c\big)\nabla_{\theta_a}\pi(s_i, \theta_a) \tag{2.52}$$

$$\text{Where} \quad \theta^T = \text{target network parameters}$$

A noteworthy achievement of the DDPG algorithm is that it was able to yield a useful reinforcement learning agent in spite of being prone to the deadly triad; DDPG uses function approximations, it updates the critic-network through a TD like method, and it is an off-policy method due to using experience replay for training. Demonstrating superior performance than DPG on which it was based [88].

Twin Delayed DDPG

One problem with DDPG is shown to be inherited from value-based methods by Fujimoto et al., and this is the over-optimism of value function estimates in the critic-network. Within the realm of value-based research, this issue has already been addressed through the introduction of double learning by van Hasselt [74], with duplicate value networks. This is not to be confused with the duplicate networks of

the target-network solution which deliberately slows down training of the target network, while double training randomly selects one of the duplicate networks for training.

Fujimoto et al. thus introduced double learning to DDPG by training two critic-networks at the same time, resulting in the Twin Delayed Deep Deterministic policy gradient (TD3) algorithm. In addition to using double learning, TD3 also improved over DDPG in two more ways. Firstly, the updates to the actor and critic are made asynchronous, with the value estimates of the critic being updated at a higher rate than the actor's policy, which mitigated the occurrence of divergent updates to policy parameters and constitutes the *delayed* part in the name of TD3.

The second improvement is to add a bit of noise to the critic learning. In DDPG the critic is trained using a TD learning approach with a learning target being obtained using the deterministic actor, introducing this determinism into the learning target can cause value-learning to suffer from high variance. Thus to combat this variance, the learning target for the critic is instead not picked using only the deterministic action from the actor, but with some noise added to the action before the learning target is retrieved from the critic. Which can be done by changing the formulation of Equation 2.51 to add some clipped random variable ϵ, which for example can be distributed normally $\epsilon \sim clip(\mathcal{N}(0, 0.1), -c, c)$, resulting in the reformulated learning target y_i being:

$$y_i = r_i + \gamma Q\big(s_i, \pi(s_{i+1}, \theta_a^T) + \epsilon, \theta_c^T\big) \tag{2.53}$$

Soft Actor Critic

Around the same time but independent of the efforts of Fujimoto et al., Haarnoja et al. [89] posed an alternative set of improvements to DDPG which culminated in the Soft Actor Critic (SAC) algorithm.

As opposed to the algorithms discussed thus far, SAC uses the unique goal maximizing return and randomness simultaneously, which is an approach called *maximum entropy reinforcement learning* [90]. In this framework, the optimality of a policy is measured not solely by the expected return, but by an expectation over the sum of return and entropy:

$$J(\pi) = \sum_t \mathbb{E}[r_t + \underbrace{\alpha \mathcal{H}(\pi(s_t))}_{\substack{\text{entropy of} \\ \text{policy}}}] \tag{2.54}$$

Where α is a temperature parameter that weighs how important being random is, and the entropy function $\mathcal{H}$ measures how random a policy is. Formulating such a loss function leads to the agent seeking to learn the most rewarding policy which is also most stochastic, Such a policy is crucial in the early stages of training, as it allows an agent to explore state and actions more widely thanks to the loss function favouring random actions, which can increase the odds of finding a near-globally optimal policy. Logically, if a deterministic policy was used, then this improvement would not have contributed to any changes in the algorithm's behaviour. Thus, Haarnoja et al. returned to using stochastic policies for the actor-network, in SAC the policy is modelled as a multivariate but diagonal Gaussian, with the actor-network outputs being the Gaussian's mean and covariance.

In addition to adopting a maximum entropy framework to evaluate its policy, SAC also uses several of the improvements present in TD3. This includes the double action-value learning trick introduced by van Hasselt [74], where just like in TD3 two action values are trained simultaneously, and a less optimistic value estimate is taken by sampling the minimum action value from the two critics when calculating the learning target for a given state-action pair.

2.4. Flight Control by Reinforcement Learning

In recent years, various deep reinforcement learning algorithms have been applied to the task of flight control. These algorithms have been used to train flight controllers for a number of aircraft types, including fixed-wing aircraft, quadcopters, and helicopters.

2.4.1. Flight Control as an MDP

The goal of this thesis is to develop an intelligent flight control system for the PH-LAB, thus it is important to understand how the flight control problem is formulated, and how that can be translated into a problem that a reinforcement learning agent can handle.

Formulating the MDP

The flight control environment is constituted of two components: the flight dynamics, and the reference trajectory. The flight dynamics component receives the action from the agent and computes how the aircraft states evolved, and then the target state that the aircraft should have is obtained from the reference trajectory. To arrive at the states and rewards of the MDP, this target and actual state are combined to produce the process's states and rewards. The specific way in which they are combined depends on the design of the MDP.

As an example, in Dally and van Kampen's work [3], the states x of the aircraft are shown in Equation 2.55, and the reference trajectory adopted defines targets for the sideslip, pitch, and roll states, as shown in Equation 2.56. In this work, two reinforcement learning controllers are implemented, meaning there are two MDPs to be defined here, for simplicity's sake this explanation will concern itself with only one controller: the attitude controller. To define the MDP states for this agent, the aircraft states x and reference trajectory x_{ref} are first combined to produce an error vector e, defined in Equation 2.57. This error vector is then weighted producing e_w, and concatenated with the aircraft's current control surface deflections and aircraft attitude rates, to produce the MDP states s shown in Equation 2.58. The inclusion of the control surface deflections may seem redundant, however, it was necessary in the case of this controller due to the agent only providing *increments* to the control surface deflections; thus providing the agent with knowledge of what current deflections are, gave more context on what increments should be fed back to the aircraft. The reward r of this MDP is then created by using e_w, which is clipped to $[-1, 0]$, the norm of it taken and scaled, as shown in Equation 2.59.

$$x = \begin{bmatrix} p & q & r & V & \alpha & \beta & \theta & \phi & \psi & h \end{bmatrix}^\top \tag{2.55}$$

$$x_{ref} = \begin{bmatrix} \beta_{ref} & \theta_{ref} & \psi_{ref} \end{bmatrix}^\top \tag{2.56}$$

$$e = \begin{bmatrix} \beta_{ref} - \beta & \theta_{ref} - \theta & \psi_{ref} - \psi \end{bmatrix}^\top \tag{2.57}$$

$$s = \begin{bmatrix} e_w^\top & \delta_a & \delta_e & \delta_r & p & q & r \end{bmatrix}^\top \tag{2.58}$$

$$r = -\frac{1}{3}\|\text{clip}(e_w, -1, 0)\| \tag{2.59}$$

Modelling Flight Dynamics

A flight dynamics model in its general form is modelled as two systems of ordinary differential equations which are functions of a vector of states x and a vector of inputs u. The first system models the derivative of the states $\dot{x}$ at any given time for the given state and inputs using the state transition functions $f(\cdot)$, whereas the second system models observations y using the observation equations $g(\cdot)$, which is how states are observed from the aircraft in a real-life system where the states are not necessarily directly known. Such a representation is shown in Equation 2.60

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u) \end{aligned} \tag{2.60}$$

These differential equations describe the motion of an aircraft in three-dimensional continuous space, but the motion has six degrees of freedom where three degrees belong to the translational motions and the remainder for rotational motions. The states of these equations typically consist of the aircraft's position p, velocity v, attitude a, and attitude rates $\mathbf{\Omega}$:

$$x = \begin{bmatrix} pos & v & a & \Omega \end{bmatrix}^T$$

$$\text{Where} \quad pos = \begin{bmatrix} x & y & z \end{bmatrix}^\top \quad v = \begin{bmatrix} u & v & w \end{bmatrix}^\top \quad a = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^\top \quad \Omega = \begin{bmatrix} p & q & r \end{bmatrix}^\top$$

The inputs vector u typically consists of the control surface deflections $\delta_a, \delta_e, \delta_r$, thrust settings T. They are typically bounded by physical limits, which are referred to as saturation limits:

$$\begin{aligned} & u = \begin{bmatrix} \delta_a & \delta_e & \delta_r & T \end{bmatrix}^\top \\ \text{Subject to} \quad & u_{min} \leq u \leq u_{max} \end{aligned}$$

This way of modelling system dynamics is called a *state-space* representation, which is signified by the use of first-order differential equations to model the evolution of states over time. The state-space representation of an aircraft can be formulated using nonlinear dynamics, which is implied when the state transition functions are denoted using lowercase f and are a function of states and/or inputs. It can also be modelled using fully linear dynamics, which is referred to as a Linear Time-invariant (LTI) model when the model parameters stay fixed over time, where the system is denoted in the following manner:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{2.61}$$

Where A, B, C, D are referred to as the state transition matrix, the input matrix, the output matrix, and the feedforward matrix respectively. LTI serves as a very useful tool, as it allows for many standard flight control design and evaluation practices to be employed. For example, they allow robust control techniques to be readily applied as the theory of $\mathcal{H}_\infty$ synthesis is founded on theory applicable only to linear systems, they allow for feedback gains to be readily calculated and thus for feedback controllers to be quickly developed, and they allow for the stability of the aircraft to be quickly analyzed by simple computation of the eigenvalues of the system [91]. A time-varying version of the linear state-space model is used in the incremental ACD case, where the A, B matrices would have different parameters over time.

To make these models useful for computing the states of the aircraft needed for an MDP, system state derivatives are integrated to compute the states in the next time step. This procedure can be simplified and a discrete version of Equation 2.60 obtained, defining the states and observations of the next time step:

$$\begin{aligned} x_{t+1} &= f(x_t, u_t) \\ y_t &= g(x_t, u_t) \end{aligned} \tag{2.62}$$

Finally, the deterministic form shown in Equation 2.62 is Markovian, as the states and inputs from the current timestep t are enough to predict the states and observations for the next timestep $t + 1$. The Markovian property remains even when stochasticity is introduced as long as no time correlation is present in this noise, methods do nonetheless exist that allow pseudo-time-correlated noise to be introduced to the model while remaining Markovian [92].

2.4.2. Learning to Fly

Flight control poses a formidable environment for reinforcement learning agents to excel in. The level of difficulty varies between aircraft designs, in agile aircraft such as fighter planes, the dynamics of such vehicles are designed to be highly unstable [93]–[95] in order to perform fast manoeuvres for the least amount of input, but as a result will be demanding to control for pilots. In contrast, commercial airliners are designed to be stable and easy to control [96], [97], and thus offer a less harsh learning environment for an agent. Additionally, the coupled nature between control actions in one axis and state changes in a separate axis makes the control problem a high-dimensional and non-linear one, which adds to the challenge for reinforcement learning agents to learn to control an aircraft.

DRL Focused Research in RL for Flight Control

One of the earliest works to apply reinforcement learning to flight control was by Abbeel et al. [98], who formulated an LQR problem for the task of acrobatic helicopter flights and used a specific instance of

reinforcement learning named differential dynamic programming to solve for the posed LQR's optimal policy. The result was a controller which can perform acrobatic manoeuvres such as flips and rolls, manoeuvres that are challenging even for human pilots.

DDPG appears to be a popular DRL algorithm applied to flight control. Fei et al. [99]managed to apply DDPG, specifically a benchmark implementation by Duan et al. [100][1], to the flight control of a flapping wing robot. This DDPG agent was trained to copy the extreme manoeuvre which hummingbirds can execute during fast escapes and managed to replicate the manoeuvre. DDPG was also adopted by De Marco et al. [101] in their research into flight control for an F-16 in a flight simulation, here the agent is trained to successfully fly an F-16 in a sequence of agile turns and manoeuvres with highly coupled dynamics, under the presence of sensor noise. The trained agent was duplicated and placed in a simulation of a prey-chaser scenario, where a prey agent was given a sequence of waypoints to follow, and a chaser agent was given the task of catching up to the prey, showing an interesting case of multi-agent interaction. Screenshots of this scenario are shown in Figure 2.6.

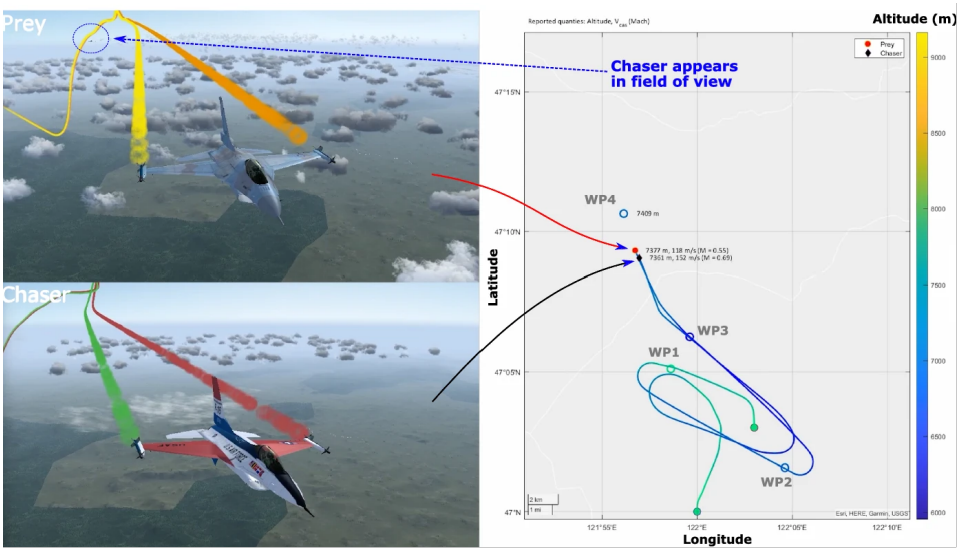


Figure 2.6: F-16 chaser simulation with DDPG pilots, screenshots of the jets in flight and their flight paths. Taken from [101]

This algorithm was also applied to the autonomous landing of fixed-wing aircraft in [102] and longitudinal control augmentation in [103].

The improved version of DDPG, namely SAC, also demonstrated success when applied to flight control. Dally and van Kampen developed a flight controller for a Cessna Citation 500 aircraft trained using SAC, alluded to in Section 2.4.1. The control structure used in this research was cascaded, while the overall controller controlled the aircraft's altitude, roll, and sideslip angle, the task was delegated to an inner and outer controller. The outer SAC controller handled altitude control by taking in reference altitude and providing a reference pitch angle for the subsequent inner SAC controller, which handled attitude control by taking in the reference pitch, roll, and sideslip, and providing control surface deflection increment commands to the aircraft. The SAC controller proved to be robust and fault-tolerant, wherein it remained performant in the face of gust disturbances, jammed or reduced effectiveness control surfaces, and loss of the horizontal tailplane. This research was followed by various efforts further supporting the ability of DRL based controllers at flight control. Such as the works of Seres et al. [104] on a Distributional SAC (DSAC) based controller, which is the distributional extension of an SAC algorithm. This algorithm remedied some of the obstacles that were observed by Dally and van Kampen in the training of an SAC flight controller. Specifically, SAC was found to provide much better training stability than SAC, showing that during training DSAC produced lower variance across different training runs at an earlier point than SAC.

The main focus of this present research is on studying how to improve the fault tolerance of flight controllers through reinforcement learning methods. To this end, the works of Dally and van Kampen

[1] Implementations available at `https://github.com/rlworkgroup/garage`

[3] show promise for continuing the development of SAC. Interestingly, research from Zahmatkesh et al. [105] shows that some degree of fault tolerance can also be achieved when coupling a discrete Q-learning algorithm with fuzzy logic, a solution which was proposed to overcome the curse of dimensionality and bridge the gap between continuous state-action space and discrete Q function domain.

ADP Focused Research in RL for Flight Control

Parallel to DRL approaches, there are also many efforts focusing on the use of ADP and ACD algorithms for the task of flight control, which possess a distinct difference in how they handle control than DRL algorithms. As mentioned in Section 2.2.4, ACD algorithms are sample efficient enough for their parameters to converge within one training episode, this is notably true for incremental-ACD algorithms such as the IDHP [52].

Enns and Si [106] produced work on using neural dynamic methods for control of a helicopter and trimming of its dynamics. This work is likely to be the first work where ADP was applied to continuous time uncertain systems, and demonstrated that it is feasible to apply ADP techniques to flight control of realistic systems in realisitic control problems. In the following year, Stengel and Ferrari [107] presented a method for training an ADP based controller using gain scheduled PID controllers obtained a-priori in an offline phase, in addition to continually training the controller in an online phase using DHP. This work leveraged optimal control theory heavily to produce an algorithm which does has less reliance on more heuristic methods such as neural network optimization with gradient descent, and brings more theoretical guarantees to the optimality of the critic function in the offline trained controller.

Li et al. [108] demonstrated that a combination of IDHP and nonlinear dynamic inversion techniques to control a tailless aircraft's attitude can quickly adapt its critic and actor parameters after a sudden deformation of the wings and damage to elevators, maintaining high accuracy in its tracking performance before and after the onset of faults. In purely IDHP implementations, the performance of the algorithm is promising in simple systems, for example in Zhou et al. [52] where a nonlinear short-period longitudinal model of an aircraft is used as the MDP environment, here IDHP was able to continue tracking a sinusoidal reference angle of attack despite the dynamic coefficients of the system changing signs during testing, which represents an effective inversion of the aircraft dynamics.

When a pure IDHP controller is applied to more complicated systems such as a full six degrees of freedom simulation and a more difficult control task such as altitude tracking, where delays between actuator input and state response are more delayed, the actor and critic would at times diverge resulting in erratic control of the aircraft. Such observations were demonstrated by Lee and van Kampen [109], where an IDHP agent served as both the outer and inner loop controller in an altitude tracking task. This occasional non-convergence can be tackled through the use of target training networks, a concept borrowed from DQN which allows parameter of actor and critic-networks to learn in a more stable manner, and has been implemented by Heyer et al. [4] to significantly improve convergence rate. The implementation from Heyer et al. is, however, not purely IDHP, as outer loop control is performed with a PID controller instead, thus it remains a question as to how well target networks may help stabilize a pure IDHP controller. Besides improving stability of the online ACD controller, Teirlinck and van Kampen [110] showed that it is possible to merge the DRL and ACD approaches in one controller, where the DRL algorithm in the form of SAC provided a robust controller on which IDHP could be augmented, to reap the benefits of both the high generalization power of DRL control and online adaptive power of ACD. This resulted in a hybrid controller that demonstrated improved fault tolerance over a purely SAC based controller and lowered divergence than a pure IDHP controller.

For simpler tasks such as rate control, where state response exhibits a much quicker reaction to actuator inputs, ADP based controllers can provide very good tracking accuracy even in the presence of noise and imperfect observations [111]. This controller was further developed and evaluated in flight tests on a Cessna Citation aircraft, where it was able to successfully perform roll and pitch rate tracking.

Research Gaps

Regarding the topic of fault tolerance. Much work has been dedicated to improving and demonstrating the robust flight qualities which DRL can provide, a consequence of being able to train the agents in an offline and safe environment for extended durations. For instance, the SAC based altitude, pitch, and sideslip tracking controller by Dally and van Kampen were given approximately 10^6 timesteps of training each at a step size of $0.01s$ before controller evaluation, corresponding to roughly 2.8 hours

of flight experience or 500 training episodes each containing $20s$ of flight time. Whereas ADP based flight controllers can converge towards an optimal control policy well within the duration of a $60s$ flight [4]. Furthermore, there remain unexplored ideas for extending ACD algorithms. In early reinforcement learning algorithms such as the TD(λ) by Sutton [78], and in more recent ADP research [35], [36], [38], [112], [113], the idea of using more than one transition or timestep's worth of information to perform actor or critic optimization allowed algorithms to be created which spanned the spectrum of using TD-like learning targets, or using MC-like learning targets, resulting in a degree of freedom that gave designers more ability to optimize performance of algorithms. Such ideas revolved around either explicitly using multiple timesteps of information to perform network updates, such as in multi-step ADP methods [35], [36], or eligibility trace updates [112], [113]. These ideas have not been applied to the case of IDHP algorithms just yet, and it remains an open but interesting question whether they can work under the incremental ACD framework, or better yet yield more optimal algorithms.

2.5. Synopsis

This chapter presented the literature and ideas that will form the basis of the present research and helped to answer some of the research questions that are posed.

To understand the field of reinforcement learning, this chapter surveyed the various classes of algorithms that are present in reinforcement learning, such as incremental-ACD algorithms and actor-critic algorithms, as well as the techniques developed to push the performance of algorithms further. For instance, one of the main issues related to reinforcement learning algorithms, in both DRL and ADP cases is learning stability of actor and or critic-networks, and a solution that is effective in both is the use of target networks. Furthermore, algorithms can be made more advanced by combining previously developed algorithms with novel augmentations, as is done in the case of Rainbow DQN. A tree diagram summarizing various reinforcement learning algorithms, from the basic to state-of-the-art is summarized in Figure 2.7. With these findings, **question 1a** of the stated research questions is answered.

From the works of Dally and van Kampen, Teirlinck and van Kampen, and various others, inspiration for how fault tolerance is defined and tested can be gathered. For instance, a controller may be called fault tolerant if tracking performance is maintained after the onset of faults, by measuring the mean squared error between reference and actual aircraft states. To perform such a test, the dynamics of the modelled aircraft may be varied to reflect a corresponding failure mode, such as reducing the control derivative of ailerons to reduce damaged aileron surfaces. Thus, **question 1b** is answered.

A survey of the same research also provided insight into how various algorithms perform when it comes to fault tolerance and tracking performance. While it's difficult to definitively say what algorithm has the absolute best fault tolerance, it is noted that SAC controllers are shown to have handled a wider array of faults than IDHP controllers, but IDHP showed that the ability to vary controller parameters mid-flight is an invaluable advantage in providing fault tolerance, while both classes of algorithms overall demonstrating comparable tracking accuracy in complex control tasks. These findings help answer **question 1c, 1d**.

Finally, several potential avenues for augmenting reinforcement learning algorithms were discovered. The two main augmentations were eligibility traces, and multi-step updates. From literature, it was identified that incorporating existing algorithms with such augmentations can improve the sample efficiency and rate at which agents converge to good policies and value functions. Therefore, these two augmentations will be further researched and incorporated into IDHP, thus answering **question 1e**.

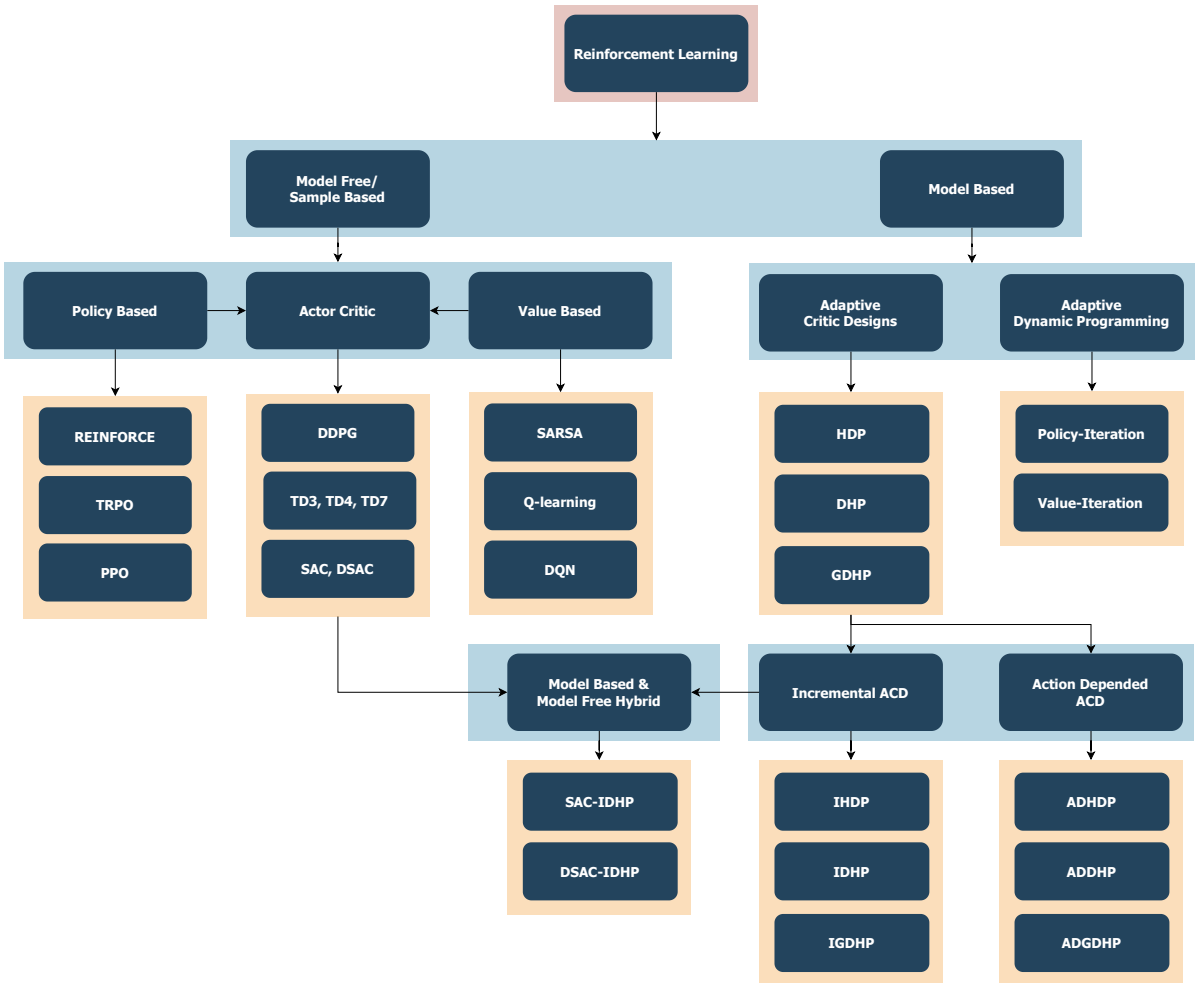


Figure 2.7: Overview of various reinforcement learning algorithms that have been encountered when compiling this literature study, light blue box presents the class of algorithms, light orange box presents reinforcement learning algorithms.

3

Preliminary Results

This chapter proposes two augmentations to be made to the IDHP reinforcement learning algorithm and presents results on how they influence the behaviour of a IDHP based flight controller, in terms of in its fault-tolerance characteristics and tracking performance.

First, the MDP on which experiments are conducted is introduced and defined in Section 3.1. Second, the IDHP algorithm which was presented in Chapter 2 will be elaborated on and described in detail in Section 3.2. Third, the augmentations proposed to be made to the IDHP algorithm are to be explained in Section 3.3. These two sections are followed by Section 3.4 which defines the IDHP algorithm itself through Algorithm 5. The chapter is then concluded by Section 3.6 which presents the results gathered from experimenting with the proposed augmentations, in addition to discussions on note-worthy results; then Section 3.7 will succeed the results and discussion section to provide a conclusion of the observed results from the experiments.

3.1. Markov Decision Process Definition

The reinforcement learning agents will be tested on a simple MDP which has few states, few actions, and simple dynamics to evaluate and compare their performances. By choosing a simpler MDP, the runtime of simulations can be shorter. But more importantly, the difficulty of implementing a minimal working example of the agents will be minimized, as more developmental effort can be diverted away from implementing the environment: which would be less demanding due to its simplicity. This is beneficial for rapid prototyping and fine-tuning, making it easier to make preliminary judgements. This MDP is centred around a flight control problem, and its details will be explained in the remainder of this section.

3.1.1. Aircraft Model and Control Task

The system on which the algorithms are to be applied is the TUDelft NLR research aircraft PH-LAB, a Cessna Citation II single-aisle business jet modified for flight testing and research purposes. The Control and Simulations group of TUDelft has carried out dynamics modelling of this aircraft. Subsequently, these dynamics are linearized and reduced into an LTI model of the short-period motion of the aircraft.

While there are many ways in which aircraft flight dynamics can be simplified each resulting in different models, the short-period motion simplifications are chosen primarily for consistency with other research done on reinforcement learning for flight control within the Control and Simulations group.

To obtain this model, the nonlinear 6-DOF flight dynamics are first linearized and decoupled into the asymmetric and symmetric motions, under the assumption that the dynamics of the states in one motion are unaffected by the second motion and vice-versa. The asymmetric motions are discarded, and the symmetric motions are further simplified to capture only the short-period eigenmotion of an aircraft. This is done using the following two assumptions:

- Airspeed remains constant throughout the eigenmotion, hence airspeed dynamics and thrust input can be omitted.
- Flight path of the aircraft remains level throughout the eigenmotion, hence pitch dynamics θ can be omitted.

This results in a simplified model containing only α and q states with elevator deflection δ_e as sole input.

This LTI model is shown in Equation 3.1, the definition of the coefficients are shown in Equation 3.2. To obtain the value of these coefficients, it is neccessary to first linearize the nonlinear dynamics of the PH-LAB model around a chosen operating point, and subsequently calculate these control and stability derivatives. The operating point which the nonlinear model is linearized around is at steady cruising condition, and the values of the resulting coefficients are presented in Table 3.1.

$$\begin{aligned} \dot{x} &= Ax + Ba \\ \Rightarrow \begin{bmatrix} \dot{\alpha} \\ \dot{q} \end{bmatrix} &= \begin{bmatrix} z_\alpha & z_q \\ m_\alpha & m_q \end{bmatrix} \begin{bmatrix} \alpha \\ q \end{bmatrix} + \begin{bmatrix} z_{\delta_e} \\ m_{\delta_e} \end{bmatrix} \delta_e \end{aligned} \tag{3.1}$$

$$\begin{aligned} z_\alpha &= \frac{V}{\bar{c}} \frac{C_{Z_\alpha}}{2\mu_c - C_{Z_{\dot{\alpha}}}} & m_\alpha &= \frac{V^2}{\bar{c}^2} \frac{C_{m_\alpha} + C_{Z_\alpha} \frac{C_{m_{\dot{\alpha}}}}{2\mu_C - C_{Z_{\dot{\alpha}}}}}{2\mu_c K_Y^2} \\ z_q &= \frac{2\mu_c + C_{Z_q}}{2\mu_c - C_{Z_{\dot{\alpha}}}} & m_q &= \frac{V}{\bar{c}} \frac{C_{m_q} + C_{m_{\dot{\alpha}}} \frac{2\mu_c + C_{Z_q}}{2\mu_C - C_{Z_{\dot{\alpha}}}}}{2\mu_c K_Y^2} \\ z_{\delta_e} &= \frac{V}{\bar{c}} \frac{C_{Z_{\delta_e}}}{2\mu_c - C_{Z_{\dot{\alpha}}}} & m_{\delta_e} &= \frac{V^2}{\bar{c}^2} \frac{C_{m_{\delta_e}} + C_{Z_{\delta_e}} \frac{C_{m_{\dot{\alpha}}}}{2\mu_C - C_{Z_{\dot{\alpha}}}}}{2\mu_c K_Y^2} \end{aligned} \tag{3.2}$$

Table 3.1: Stability and control derivatives for the PH-LAB at cruise condition.

V	$=$	$59.9m/s$	C_{Z_α}	$=$	-5.16	C_{m_α}	$=$	-0.43
$\bar{c}$	$=$	$2.022m$	$C_{Z_{\dot{\alpha}}}$	$=$	-1.43	$C_{m_{\dot{\alpha}}}$	$=$	-3.7
μ_c	$=$	102.7	C_Z	$=$	-3.86	C_{m_q}	$=$	-7.04
K_Y^2	$=$	0.98	$C_{Z_{\delta_e}}$	$=$	-0.6238	$C_{m_{\delta_e}}$	$=$	-1.553

This short period model's dynamics are considered representative for relatively short durations of simulations, since the model is derived based on the assumptions of the short-period response. Over longer simulation durations, the match between this model's states and the nonlinear or a real PH-LAB's states will begin to diverge, as the phugoid motion of the real aircraft will not manifest in the simulations. Nonetheless, this model is considered sufficient to provide the reinforcement learning agents with an environment which is adequately realistic, striking a good balance between fidelity and simplicity.

Lastly, the control task of this problem is to be an Angle of Attack (AoA) reference tracking task, where the IDHP agent has to learn to control the aircraft's AoA to follow a reference AoA signal. The design of this reference signal will be detailed in Section 3.5.

3.1.2. MDP Environment Specification

While the aircraft is what the agents ultimately control, they are to only interface with the MDP environment. The consequence of this is that the state and action space of the environment are not necessarily equivalent to those of the aircraft model, in this case, the short-period LTI. The MDP environment will be specified in this subsection, this entails defining the environment state s, action a, and reward r spaces.

For the LTI, the state and action spaces are denoted x and u, with their definitions given in Equation 3.3, from where it can be seen that the model state space is two dimensional and the input space is one dimensional.

$$x = \begin{bmatrix} \alpha \\ q \end{bmatrix} \qquad u = \delta_e \tag{3.3}$$

For the MDP environment, the state and action space definitions are design variables, they can be freely defined as long as they are related to the control problem at hand. Logically, the goal in these design decisions is to allow for the best-performing agents to be achieved, which involves consideration of several factors. For example, the state space should be designed such that the MDP is Markov, this theoretically gives the agents sufficient information about the environment to provide ideal performance. Luckily in the case of the short-period LTI, finding a Markov MDP environment is trivial as LTI state space models are, by definition, Markov. Furthermore, the scale of variables in s deserves consideration, as it is generally desirable for inputs to neural networks to belong in the same orders of magnitude if not entirely normalized [114].

Regarding s for the present MDP, some options are presented in the following:

$$s = \begin{cases} \text{Option 1:} & \alpha_{ref} - \alpha \\ \text{Option 2:} & \begin{bmatrix} \alpha & \alpha_{ref} \end{bmatrix}^\top \\ \text{Option 3:} & \begin{bmatrix} q & \alpha & \alpha_{ref} \end{bmatrix}^\top \\ \text{Option 4:} & \begin{bmatrix} q & \alpha & \alpha_{ref} - \alpha \end{bmatrix}^\top \end{cases} \tag{3.4}$$

These options are evaluated in an initial testing phase by running a Monte Carlo simulation, where the IDHP agent is run on each of the 4 options on 15 episodes, with the same IDHP hyperparameters used across all options. Using the “eye norm”[115] on the tracking performance in these simulations, it was determined that **Option 1** of a single dimensional MDP state space with the tracking error being the state variable, gave the best result.

For a, two possible options are to directly control the elevator deflection, or to control the increments to the elevator deflection:

$$a = \begin{cases} \text{Option 1:} & \delta_e \\ \text{Option 2:} & \Delta\delta_e \end{cases} \tag{3.5}$$

Similar to the state options, these two options are evaluated through Monte Carlo simulations to see which one allows for better tracking performance, where it was found that **Option 1** of a single dimensional MDP action space with the variable being elevator deflection angle gave the best results.

The reward signal is defined as a squared tracking error multiplied by a reward scaling factor κ, as shown in Equation 3.6. An associated reward gradient is defined as well, in Equation 3.7.

$$r = \frac{\kappa}{2}(\alpha - \alpha_{ref})^2 \tag{3.6}$$

$$\frac{\partial r}{\partial x} = \kappa \begin{bmatrix} \alpha - \alpha_{ref} & 0 \end{bmatrix}^\top \tag{3.7}$$

Lastly, one episode will last 60 s of simulation time, with a time step duration of 0.02 s.

3.1.3. Fault Scenarios

Several flight scenarios are designed to test the fault tolerance of the agent during some of the experiments. This is done by testing the agent on three different events: shifted centre of gravity, decrease in elevator effectiveness, and reversed elevator controls. The first two cases emulate faults which can be realistically expected to happen. For instance, a shift in the centre of gravity can occur if payloads are moved around in the aircraft. A decrease in elevator effectiveness can occur on catastrophic events, such as shrapnels from engine explosion, or perhaps excessive nose-up during takeoff damaging the tail. The last of the scenarios is not expected to be realistic, as there are unlikely to be events that cause elevator controls to suddenly be reversed; nonetheless, this fault can demonstrate the adaptiveness of the implemented agents to dramatic changes in system dynamics, which in turn implies fault tolerance.

Shifted Centre of Gravity

In this scenario, the centre of gravity is shifted from the nominal position towards the aircraft nose by half a metre. To model this, the stability and control derivatives can be recomputed based on their definitions [116], whereupon the LTI matrices can be redefined. The affected derivatives are denoted by an asterisk in the exponent and updated according to the following equations, they are functions of Δx which is positive towards the tail and in metres, and are defined in the following:

$$
\begin{aligned}
C^*_{m_\alpha} &= C_{m_\alpha} - C_{Z_\alpha}\frac{\Delta x_{c.g}}{\bar{c}} \\
C^*_{m_{\dot{\alpha}}} &= C_{m_{\dot{\alpha}}} - C_{Z_{\dot{\alpha}}}\frac{\Delta x_{c.g}}{\bar{c}} \\
C^*_{Z_q} &= C_{m_q} - C_{Z_\alpha}\frac{\Delta x_{c.g}}{\bar{c}} \\
C^*_{m_q} &= C_{m_q} - (C_{Z_\alpha} + C_{m_\alpha})\frac{\Delta x_{c.g}}{\bar{c}} + C_{Z_\alpha}(\frac{\Delta x_{c.g}}{\bar{c}})^2 \\
C^*_{m_{\delta_e}} &= C_{m_{\delta_e}} - C_{Z_{\delta_e}}\frac{\Delta x_{c.g}}{\bar{c}}
\end{aligned}
$$

Damaged Elevator

In this scenario, the control authority of the elevator is to be reduced by half. This can be modelled relatively easily, by dividing the input matrix of the short-period LTI by two, which halves the elevators' control effectiveness. The faulty input matrix B^* is defined according to Equation 3.8.

$$B^* = \frac{B}{2} \tag{3.8}$$

Reversed Elevator

In the last scenario, the mechanics on the ground accidentally rewired the flight control computer, which meant that at some point in the flight, the elevators would suddenly reverse the control commands it received. This is modelled by multiplying the input matrix of the LTI by -1, resulting in the faulty input matrix B^* shown in Equation 3.9.

$$B^* = -B \tag{3.9}$$

3.1.4. MDP Summary

The MDP which the agents are tested on revolves around angle of attack control on a simple model of an aircraft. The agent will provide the elevator deflection angles at each time step to the MDP environment, which will use this input to determine the subsequent time step's state and rewards which in turn is fed back to the agent.

When the environment receives action a from the agent, it feeds a as the input u to the short-period LTI model, which returns the next time step's model states. The model states and the reference angle of attack are used to calculate the subsequent time step's MDP state and reward.

The MDP environment variables are summarized in Table 3.2, and the flow of MDP variables is visualized in Figure 3.1.

Table 3.2: MDP environment summary.

Model state: x_t	$=$	$\begin{bmatrix}\alpha_t & q_t\end{bmatrix}^\top$	MDP state: s_t	$=$	$\alpha_t - \alpha_{ref,t}$
Model or MDP action: u_t, a_t	$=$	$\delta_{e,t}$	MDP reward: r_t	$=$	$-\frac{\kappa}{2}(\alpha_t - \alpha_{ref,t})^2$
MDP reward gradient: $\frac{\partial r_t}{\partial x_t}$	$=$	$-\kappa\begin{bmatrix}(\alpha - \alpha_{ref}) & 0\end{bmatrix}^\top$			
No. model states: n	$=$	2	No. model inputs: m	$=$	1
Time step duration: dt	$=$	0.02 s	Episode duration: T	$=$	60 s

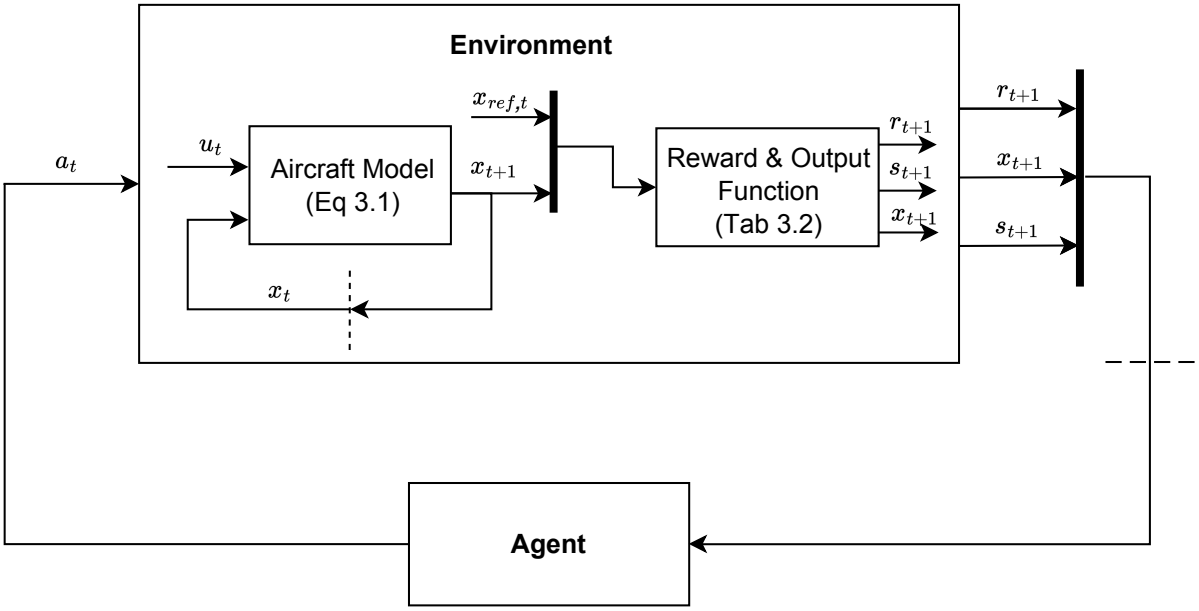


Figure 3.1: MDP environment flow diagram.

3.2. IDHP Agent

In answering research question **1**, it was identified that IDHP has the most promising fault tolerance potential. This section will introduce the details and the structure behind this reinforcement learning algorithm.

The overall algorithm consists of three components; the model, the critic, and the actor. These components are explained in Section 3.2.1, Section 3.2.2, and Section 3.2.3 respectively. The update rules thus written for the IDHP algorithm are taken from Zhou [52], with some nomenclature changes where variable names are swapped with reinforcement learning vocabulary; e.g. the use of reward instead of cost-to-go, and return function instead of loss function for the actor.

3.2.1. Model

The IDHP's model is meant to represent the system dynamics within the MDP environment, allowing it to update actor and critic parameters more efficiently. The model structure used is a discrete-time Linear Time-Varying (LTV) model, which takes in one time step's state and actions and outputs the next time step's states. This model structure can be written as Equation 3.10.

$$x_t = F_t x_{t-1} + G_t u_{t-1} \tag{3.10}$$

In IDHP, this model is identified using a RLS algorithm, specifically an exponentially weighted variant, which offers a computationally more efficient estimator than naive linear estimation for recursive estimation theoretically without numerical instability issues through using the matrix inversion lemma, a long-standing problem solved by Slock and Kailath [117]. With this algorithm, it became possible to estimate linear system models online assuming a sufficiently high sample rate, which to some degree ensures the observed dynamics are linear even when the dynamics are nonlinear over a large time scale. This advantage is leveraged to make DHP incrementally identify the system model, thus yielding the IDHP algorithm [118]. To understand and implement RLS, Haykin's work on Adaptive Filter Theory was used [119].

In RLS, the model parameters and model variables are grouped into Θ_t and X_t according to Equation 3.11 and Equation 3.12 respectively.

$$\Theta_t = \begin{bmatrix} F_t^\top \\ G_t^\top \end{bmatrix} \tag{3.11}$$

$$X_t = \begin{bmatrix} \delta x_t \\ \delta a_t \end{bmatrix} \tag{3.12}$$

$$\delta x_t = x_t - x_{t-1}$$
$$\delta a_t = a_t - a_{t-1}$$

To perform model estimation, an RLS covariance matrix Σ_t, an RLS gain k_t, an RLS prediction $\delta\hat{x}_t$, and an RLS error or innovation ϵ_t are defined according to the following definitions:

$$\Sigma_t \in \mathbb{R}^{n+m}$$
$$k_t = \frac{\Sigma_{t-1} X_t}{\rho + X_t^\top \Sigma_{t-1} X_t}$$
$$\delta\hat{x}_t = X_t^\top \Theta_{t-1}$$
$$\epsilon_t = \delta x_t - \delta\hat{x}_t$$

Where ρ is the RLS forgetting factor. The two RLS variables which are directly related to model estimation, Θ and Σ, are updated according to Equation 3.13 and Equation 3.14 respectively.

$$\Theta_t = \Theta_t + k_t \epsilon_t \tag{3.13}$$

$$\Sigma_t = \frac{1}{\rho}(\Sigma_{t-1} - k_t X_t^\top \Sigma_{t-1}) \tag{3.14}$$

One possible option for initializing these two variables is to use Equation 3.15 with σ a large number.

$$\Theta_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad \Sigma_0 = \sigma \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.15}$$

Such an initialization offers a good initial guess of what the system dynamics may be, since the discrete state transition matrix approaches the identity matrix as the sample rate approaches infinity, and the discrete state input matrix approaches zero under likewise conditions; while directing the algorithm to perform big updates on the parameter matrix through initializing the covariance matrix to be large:

$$F_t = \mathbb{I} + dtA \Rightarrow \lim_{dt \to 0} F_t = \mathbb{I}$$
$$G_t = dtB \Rightarrow \lim_{dt \to 0} G_t = \mathbf{0}$$

Where A and B are the continuous state transition and input matrices respectively, and dt is the time step size in unit time. That being said, it was observed that initializing Θ with a null matrix also provides satisfactory results.

Finally, the RLS algorithm can be found summarized in Algorithm 4.

Algorithm 4 RLS algorithm.

1: **Initialize**. Parameter covariance $\Sigma_0 \in \mathbb{R}^{(n+m)\times(n+m)}$, parameter $\Theta_0 \in \mathbb{R}^{(n+m)\times n}$.
2: **Online Model Identification**. For each time step, $t = 1, 2, \cdots$, compute:

$$\begin{aligned}
\delta x_t &= x_t - x_{t-1} \\
\delta a_t &= a_t - a_{t-1} \\
X_t &= \begin{bmatrix} \delta x_t & \delta a_t \end{bmatrix}^\top \\
k_t &= \frac{\Sigma_{t-1} X_t}{\rho + X_t^\top \Sigma_{t-1} X_t} \\
\epsilon_t &= \delta x_t - X_t^\top \Theta_{t-1} \\
\Theta_t &= \Theta_t + k_t \epsilon_t \\
\Sigma_t &= \frac{1}{\rho}(\Sigma_{t-1} - k_t X_t^\top \Sigma_{t-1})
\end{aligned}$$

3.2.2. Critic

The critic of IDHP is a value gradient function, meaning it outputs the gradient of the value function. This function uses a single-layer neural network function approximator:

$$\lambda(s; W_c) = \frac{\partial V(s)}{\partial x}$$

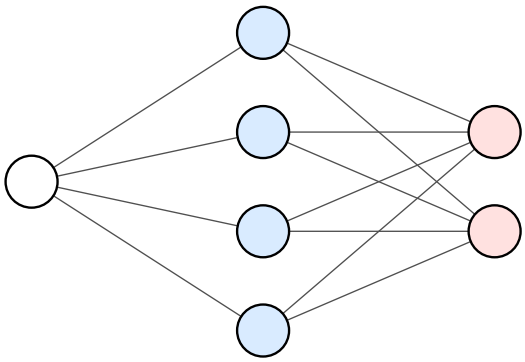
Where W_c are the network weights parametrizing the critic-network, the semicolon delimiter denotes that the critic is parameterized by W_c.

This network has an input layer, one hidden layer, and an output layer. The domain of the critic-network spans the MDP state space which only has the single variable s, as defined in Table 3.2. Thus the input layer has a single neuron, it uses a linear activation function and has no output bias. The hidden layer consists of four neurons all using the $\tanh$ activation function and no output bias term. The number of neurons in the hidden layer was decided through preliminary testing of algorithm performance using a varying number of hidden neurons, where it was found that a small number of neurons had slightly poorer but very similar performance than higher numbers, and it was decided that a small hidden layer would be interesting to investigate to see the limit of network sizes for a usable flight controller. For the output layer, since the critic outputs the gradient of a state's value w.r.t. the model states x, the output layer thus has two output neurons, one neuron for the derivative w.r.t. each model state. The output neurons use a linear activation function with no output bias. The network can also be expressed as a **vector** function, as done in Equation 3.16.

$$\begin{aligned}
\lambda(s; W_c) &= W_{c,2} \tanh(W_{c,1} s) \\
\lambda(\cdot) \in \mathbb{R}^2 \quad W_{c,1} &\in \mathbb{R}^{4\times 1} \quad W_{c,2} \in \mathbb{R}^{2\times 4}
\end{aligned} \tag{3.16}$$

Where the n in $W_{c,n}$ denote the layer number, counting from the input layer. A graphical depiction is also shown in Figure 3.3.

The critic-network is trained using temporal difference learning, by using previous estimates and current observations to improve its future estimation accuracy. This is done by defining a temporal difference error δ which quantifies the critic's accuracy, and then continuously updating critic parameters W_c to minimize the squared temporal difference error E through stochastic gradient descent. While



Input Nodes	$= 1$	Hidden Nodes	$= 4$	Output Nodes	$= 2$
Activation	$= linear$	Activation	$= tanh$	Activation	$= linear$
Bias	= None	Bias	= None	Bias	= None

Figure 3.2: IDHP critic-network.

theoretically, more advanced gradient descent algorithms such as AdaGrad or Adam can be used instead of simple stochastic gradient descent, experiments have shown that such optimizers can slow down learning of temporal difference methods, this is discussed by Gupta [120] and Nichols [121], and will be elaborated on in Section 3.3.2.

The IDHP critic δ and the quadratic temporal difference error E are formulated in Equation 3.18 and Equation 3.17 respectively.

$$E_t = \frac{1}{2}\delta_t^\top \delta_t \tag{3.17}$$

$$\delta_t = \lambda_{t-1} - \frac{\partial r_{t-1}}{\partial x_{t-1}} - \gamma\lambda'_t \left.\frac{\partial x_t}{\partial x_{t-1}}\right|_t \tag{3.18}$$

$$\lambda_{t-1} = \lambda(s_{t-1}; W_c(t-1)) \qquad \lambda'_t = \lambda'(s_t; W_{c'}(t-1))$$

$$\left.\frac{\partial x_t}{\partial x_{t-1}}\right|_t = F_t + G_t \frac{\partial a_{t-1}}{\partial x_{t-1}} \tag{3.19}$$

Where the vertical bar in $\left.\frac{\partial x_t}{\partial x_{t-1}}\right|_t$ denotes the time step for which this term is evaluated, i.e. the model matrices F_t and G_t are to be found at timestep t.

Note that in formulating δ_t, a target critic λ' is used to calculate the value gradient of the s_{t+1}. This target critic is identical to the critic, however, it is not trained but simply updated at a slower rate than the critic, and serves to stabilize the learning process of the critic-network by reducing the variance of target value gradients [4], [122].

The gradient descent equation used to optimize the critic for minimal E involves determining the gradient of E w.r.t. critic-network weights and incrementing critic weights by some fraction η_c of the negative of that gradient, the negative here is necessary as this is a minimization problem. This procedure can be summarized by Equation 3.20. To update the target critic, a moving average filter is used to trickle the latest critic weights into the target critic, shown in Equation 3.21.

$$W_{c,t} = W_{c,t-1} - \eta_c \frac{\partial E_{t-1}}{\partial W_c} \tag{3.20}$$

$$W_{c',t} = \tau W_{c,t-1} + (1-\tau) W_{c',t-1} \tag{3.21}$$

With the t subscript in $W_{c,t}$ denoting the time step which the weights are taken, and where η_c is the learning rate of the critic, $\tau \ll 1$ is a mixing factor typically made a very small positive number. To be able to implement Equation 3.20 and train the critic-network, it is necessary to expand the term $\frac{\partial E(t)}{\partial W_c}$:

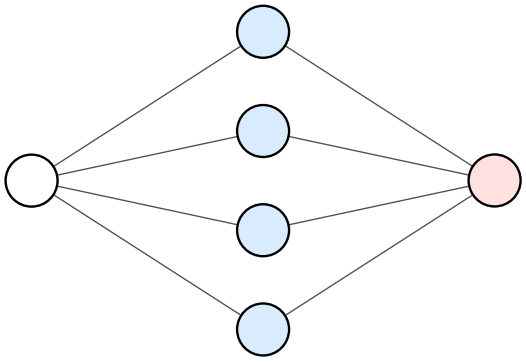
$$E_t = \frac{1}{2}\delta_t^\top \delta_t$$
$$\frac{\partial E_t}{\partial W_c} = \frac{\partial E_t}{\partial \delta_t}\frac{\partial \delta_t}{\partial \lambda(\cdot)}\frac{\partial \lambda(\cdot)}{\partial W_c}$$
$$\frac{\partial E_t}{\partial W_c} = \delta_t \frac{\partial \lambda(\cdot)}{\partial W_c}$$

Thus, the critic weights are incremented by a product of δ_t and the gradient of the critic output w.r.t. the weights. The latter partial derivative term is defined since a neural network is analytically differentiable, the most common practice of calculating this derivative is through backpropagation algorithms, which efficiently evaluates this derivative thanks to the storage of intermediate variable during a feedforward operation of a network [123].

3.2.3. Actor

The actor of IDHP is a policy function which uses a single-layer neural network as a function approximator. This network comprises an input layer, one hidden layer, and an output layer. Just like the critic, the domain of the actor-network spans the MDP state space which only has the single variable s, thus the input layer has a single neuron. The input layer uses a linear activation function with no output bias. The hidden layer consists of four neurons all using the $\tanh$ activation function and no output bias term. The number of neurons in the hidden layer was once again decided upon based on preliminary testing, where it was found four neurons gave satisfactory algorithm performance. Lastly, for the output layer, the range of the policy function is the MDP action space which only has a single action a as defined in Table 3.2, thus the output layer has a single output neuron, which uses a $\tanh$ activation function and has no output bias. The network can be expressed as a **scalar** function, as done in Equation 3.22, where $W_{a,n}$ are the network weights parametrizing the actor-network with n being the layer number, counting from the input layer. The actor-network is also depicted graphically in Figure 3.3.

$$\pi(s; W_a) = \tanh(W_{a,2}\tanh(W_{a,1}s)) \tag{3.22}$$
$$\pi(\cdot) \in \mathbb{R} \quad W_{a,1} \in \mathbb{R}^{4\times 1} \quad W_{a,2} \in \mathbb{R}^{1\times 4}$$



Input Nodes	$= 1$	Hidden Nodes	$= 4$	Output Nodes	$= 1$
Activation	$= linear$	Activation	$= tanh$	Activation	$= tanh$
Bias	$=$ None	Bias	$=$ None	Bias	$=$ None

Figure 3.3: IDHP actor-network.

Note that since the output of the actor-network passes through a $tanh$, this means the output value is bound to the range $(-1, 1)$. Since the elevator deflection limit on the Cessna Citation II is from $[-20, 20]$ deg, it is thus necessary to scale the actor output by a factor of 20 before it can be used as the input value to the aircraft model. After this scaling, the actor's action a_t is retrieved.

To train the actor, its weights are updated to maximize the return function Equation 3.23 using stochastic gradient ascent Equation 3.24.

$$R_t = r_t + \gamma J(s_{t+1}) \tag{3.23}$$

$$W_{a,t} = W_{a,t-1} + \eta_a \frac{\partial R_{t-1}}{\partial W_a} \tag{3.24}$$

With t in the subscript of $W_{a,t}$ denoting time step, η_a is the learning rate of the actor. It can be observed that the return function utilizes the state value function $J(s)$, which is not estimated in the present algorithm. This will prove to be a non-issue, as ultimately what is needed to perform actor weight updates are derivatives of $J(s)$. So, to further demonstrate this point, and for implementation of the gradient ascent update, the partial derivative $\frac{\partial R_{t-1}}{\partial W_a}$ is expanded:

$$\begin{aligned}
\frac{\partial R_{t-1}}{\partial W_a} &= \frac{\partial R_{t-1}}{\partial x_t} \frac{\partial x_t}{\partial a_{t-1}} \frac{\partial a_{t-1}}{\partial W_a} \\
&= \left[\frac{\partial r_{t-1}}{\partial x_t} + \gamma \lambda'(s_t) \right] \left. \frac{\partial x_t}{\partial a_{t-1}} \right|_t \frac{\partial a_{t-1}}{\partial W_a} \\
\text{Where} \quad \left. \frac{\partial x_t}{\partial a_{t-1}} \right|_t &= G_t
\end{aligned}$$

Observe that $J(s)$ disappears from the equation, and is replaced by $\lambda(s)$, which *is* estimated by the algorithm in the form of the critic.

Strictly speaking, the term $\frac{\partial r_{t-1}}{\partial x_t}$ is undefined, as it relates the derivative of past reward to a future model state. However, under the assumption of smooth system dynamics and high sampling rate, it is possible to replace this term with $\frac{\partial r_t}{\partial x_t}$. This re-expression is adopted in the present implementation.

3.2.4. Adaptive Learning

The actor and critic are tweaked to be more and less adaptive depending on the situation. During a warmup period and when a fault is detected in the aircraft, the agents will use a higher learning rate to be more adaptive. When the aircraft is functioning without anomaly and past the warmup period, the agents adopt a low learning rate.

The high and low learning rates are to be active under the conditions specified by Equation 3.25.

$$\eta = \begin{cases} \eta_{\cdot,h} & \text{if} \quad t < 3 \text{ or } ||e_{[t-2,t]}|| > 1 \\ \eta_{\cdot,l} & \text{otherwise} \end{cases} \tag{3.25}$$

Where t is in seconds, and $e_{[t-2,t]}$ denotes the AoA tracking error in degrees between $t-2$ and t. In words, the learning setting is set to adaptive if the simulation time is less than 3 s or if the tracking error at any point in the past 2 s is greater than 1 deg. Otherwise, the learning setting is set to normal.

When it comes to the RLS model, a known problem associated with exponentially increasing covariance occurs when the system is not persistently excited [118], resulting in dramatic changes to parameter estimates once the system is excited. To counter this, the RLS forgetting factor ρ is set to 1, meaning that the latest parameters are estimated using all previous measurements, similar to ordinary least squares. This can be troublesome since the model parameters will vary less as time goes on, hampering adaptiveness. When the simulation time has proceeded past the warmup phase, if the RLS ϵ norm exceeds 9×10^{-5}, then Σ is re-initialized to a high variance matrix identical to the initial Σ_0, and the matrix will not be reset for the next 3 s, see Equation 3.26. The same reinitialization is done for the model parameter matrix Θ, see Equation 3.27.

$$\Sigma_{t+1} = \begin{cases} \text{Reset to } \Sigma_0 & \text{if} \quad ||\epsilon_t|| > 9 \times 10^{-5} \\ \text{Update using Equation 3.14} & \text{otherwise} \end{cases} \tag{3.26}$$

$$\Theta_{t+1} = \begin{cases} \text{Reset to } \Theta_0 & \text{if} \quad ||\epsilon_t|| > 9 \times 10^{-5} \\ \text{Update using Equation 3.13} & \text{otherwise} \end{cases} \tag{3.27}$$

3.2.5. IDHP Agent Summary

The reinforcement learning agent used to solve this MDP problem is an IDHP agent. This agent receives the MDP state, MDP reward, and model state at every time step. It then uses these signals to both compute that time step's action, as well as to update it's internal variables, namely the RLS model, the actor, and the critic. A flow chart of the IDHP agent is given in Figure 3.4.

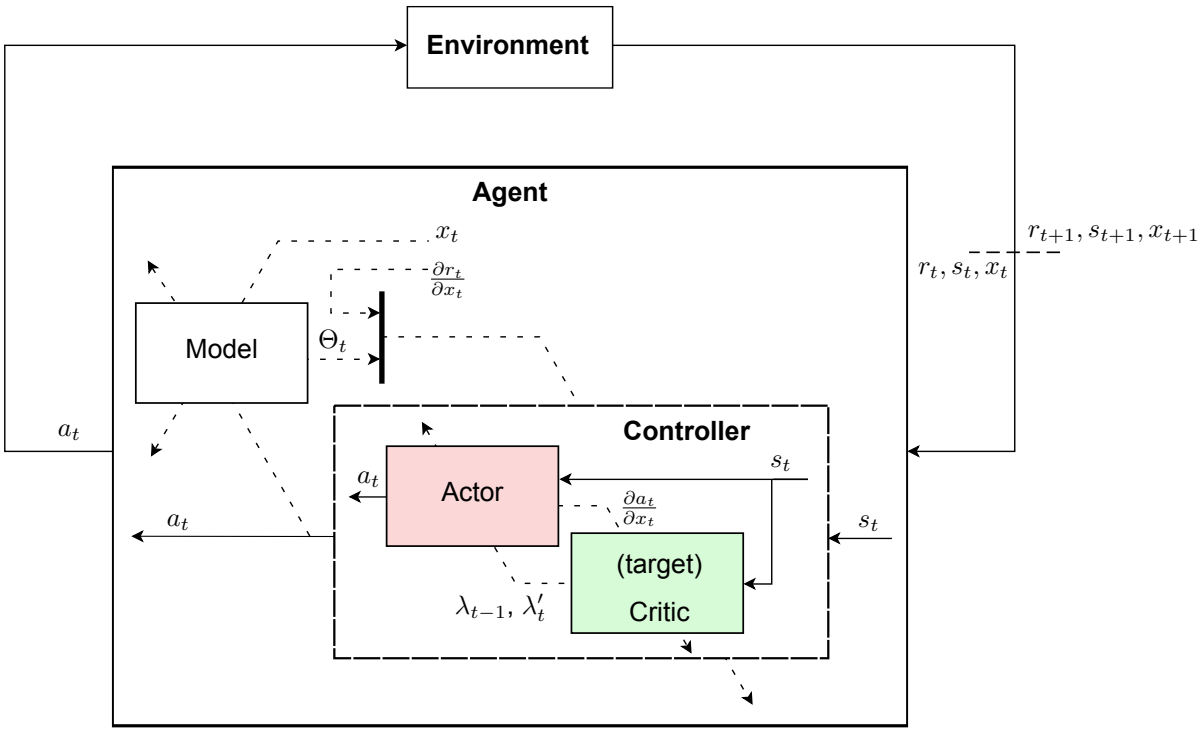


Figure 3.4: MDP agent flow diagram, dashed signals represent variables used to update the blocks which they cross.

3.3. IDHP augmentations

The IDHP algorithm is to be researched further to explore possibilities of improving it's fault tolerance and tracking performance. Potential augmentations that IDHP can leverage have been identified during the literature study phase, and they were briefly introduced in Chapter 2. The first of such augmentation is the multi-step update, explained in Section 3.3.1, and the second augmentation is the eligibility trace, explained in Section 3.3.2

3.3.1. Multistep Temporal Difference

IDHP uses a temporal difference approach in training the critic. Temporal difference methods use observed MDP state-reward pairs to compute a temporal difference target, this target constitutes the second and third term in the δ definition introduced in Equation 3.18, repeated below for clarity:

$$\delta_t = \lambda_{t-1} - \underbrace{(\frac{\partial r_{t-1}}{\partial x_{t-1}} + \gamma \lambda'_t \frac{\partial x_t}{\partial x_{t-1}})}_{\text{IDHP TD Target}}$$

The TD target is what a value function, or in the IDHP case the value gradient function, is trained to

approximate. This target is an empirical estimation of what the true value/value gradient is for the current state. Thus, in theory, the more accurate this target is the faster the estimated value function approaches the true value function.

With the works of Watkins [124] and Cichosz [125] the idea of multi-step or truncating TD was introduced and demonstrated to improve learning rates of TD algorithms. Multi-step TD, also known as n-step TD, uses state-reward pairs observed over multiple time steps to reduce bias in the TD target, this generally takes the form of Equation 3.28.

$$\text{n-step TD Target} : \gamma^n V(s_n) + \sum_{m=0}^{n-1} \gamma^m r_m \tag{3.28}$$

Where it can be seen that more than one reward observation and the $n\text{-}th$ state value estimate are used to construct the TD target.

Precedence of n-step TD returns yielding improved agent learning rates can be found for simple TD algorithms in the textbook by Sutton and Barto [17], as well as for HDP in the works by Luo et al. [35] and Wang et al. [36], who proposed a different flavour of multi-step augmentation where policy evaluation is performed for several steps as opposed to gathering state-reward over multiple time steps.

The present research will extend the multi-step idea to the IDHP algorithm. To this end, a multi-step target can be constructed and the multi-step TD error δ_n can be constructed:

$$\delta_{n,t} = \lambda_{t-n} - \frac{\partial(\gamma^n V(s_t) + \sum_{m=0}^{n-1} \gamma^m r_{t-n+m})}{\partial x_{t-n}} \tag{3.29}$$

For the sake of simplicity, the present thesis will only consider the 2-step TD error, where the TD error for the critic takes the form of Equation 3.30.

$$\delta_{2,t} = \lambda_{t-2} - (\frac{\partial r_{t-2}}{\partial x_{t-2}} + \gamma \frac{\partial r_{t-1}}{\partial x_{t-1}} \frac{\partial x_{t-1}}{\partial x_{t-2}}\bigg|_{t-1} + \gamma^2 \lambda'_t \frac{\partial x_t}{\partial x_{t-1}}\bigg|_t \frac{\partial x_{t-1}}{\partial x_{t-2}}\bigg|_{t-1}) \tag{3.30}$$

With the state transition terms $\frac{\partial x_{t-1}}{\partial x_{t-2}}\Big|_{t-1}$ and $\frac{\partial x_t}{\partial x_{t-1}}\Big|_t$ calculated according to Equation 3.19.

3.3.2. Eligibility Traces

Eligibility traces are an alternative method for incorporating more prior information to agent training, this is achieved by storing the past neural network updates in an eligibility trace $\mathbf{E}$, and using them for several subsequent updates at decaying magnitudes. This results in a new set of equations used to update the network parameters.

$$\begin{aligned} W_{t+1} &= W_t + \eta \delta_t \mathbf{E}_t \\ \mathbf{E}_{t+1} &= \lambda\gamma \mathbf{E}_t + \nabla W_t, \quad \mathbf{E}_0 = 0 \end{aligned} \tag{3.31}$$

Where ∇W is the gradient of the network output w.r.t network weights, δ_t is the gradient of the relevant metric w.r.t. network output, for the critic this metric is the TD error and for the actor it's the return; λ is the trace decay rate which controls how quickly do prior gradients decay in the eligibility trace.

The idea of extending IDHP to a multi-step version leads one to also consider the case of extending IDHP with eligibility traces as well, as the history of their inception is closely intertwined. This connection is explored in several works, one of such works is by Singh and Sutton [126] who explored the similarities of eligibility traces algorithms and MC algorithms, which are the infinite-step extreme of multi-step algorithms. Another such work is by van Seijen [127], who derived the distinction between the eligibility trace algorithm TD(λ) and the λ return algorithm which is the prototypical multi-step algorithm. In fact, van Seijen proved that the two algorithms are asymptotically identical for infinitely small learning rates.

As opposed to the multi-step augmentation, eligibility traces will only be applied to the actor update equations. This is a decision made as a result of empirical experiments which concluded that IDHP learning is more stable when only the actor uses eligibility traces.

As it turns out, eligibility traces can be formulated in several manners [128], these formulations each record eligibility traces slightly differently. For the present thesis, two options will be experimented with. They are namely the; accumulating trace, and the replacing trace, for which they are written below for the actor updates. These two trace's generic form can be obtained by swapping the partial derivative term with $\nabla f(x)$, gradient of performance metric w.r.t. function parameters.

1. Accumulating traces:
$$\mathbf{E}_{t+1} = \lambda\gamma\mathbf{E}_t + \nabla W_{a,t} \tag{3.32}$$
2. Replacing traces:
$$\mathbf{E}_{t+1} = \begin{cases} \nabla W_{a,t} & \text{if} \quad ||\nabla W_{a,t}|| > ||\lambda\gamma\mathbf{E}_{t-1}|| \\ \lambda\gamma\mathbf{E}_{t-1} & \text{otherwise} \end{cases} \tag{3.33}$$

It can be seen that the first option, accumulating traces, is simply the original form of eligibility traces shown in Equation 3.31.

These two traces have different effects on how an agent learns, which can be roughly illustrated with Figure 3.5. With accumulating traces, when one state is visited several times, the gradient update which results from the observed state-reward pair is compounded on each other, which can cause faster network weight changes. These compounded updates will be persistently made to the network for several time steps following this visit, albeit at decayed amounts. On the other hand, with replacing traces, when one state is visited several times even in quick succession, the magnitude of network weight updates will not exceed that of the original network weight gradients. However, the effect of this state-reward pair update will persist after the state visit, as the eligibility trace will continue to update the network with the observed gradient, albeit again with decaying amounts.

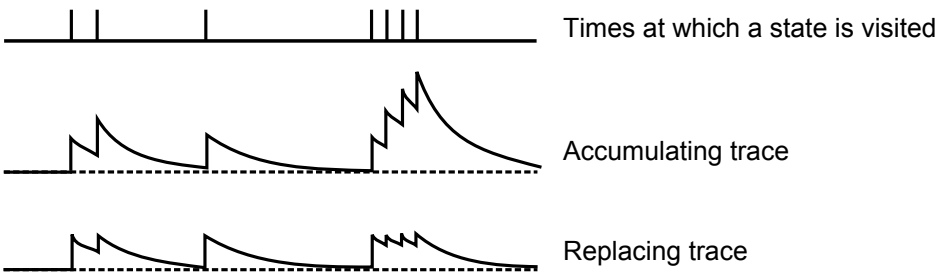


Figure 3.5: Accumulating and replacing trace illustrated, recreated from [126].

To arrive at the final network weight update procedure, the eligibility traces can be used to update actor weights according to Equation 3.31 with δ_t being the return gradient shown' in Equation 3.34.

$$\delta_{a,t} = \frac{\partial R_t}{\partial a_t} = \left[\frac{\partial r_{t-1}}{\partial x_t} + \gamma\lambda(s_t)\right] G_t \tag{3.34}$$

Eligibility Trace and Momentum Gradient Descent

An interesting note can be made about the differences between eligibility traces and momentum-based gradient descent algorithms.

The high-level ideas of the two algorithms are similar. In neural networks, momentum-based algorithms for network weight optimization roughly take the following form:

$$W_{t+1} = W_t + \eta \mathbf{M}_t \tag{3.35}$$
$$\mathbf{M}_{t+1} = \lambda \mathbf{M}_t + \delta_t \nabla W_t, \quad \mathbf{M}_0 = 0$$

The most important difference in the momentum-based update equations is the metric gradient term δ_t, which is moved from the weight update to the momentum/trace update. Note that the momentum variable $\mathbf{M}$ is equivalent to $\mathbf{M}$ in its function, they both serve to store past updates for use in future weight updates.

This difference can be used intuitively to deduce that momentum-based optimization for actor or critic optimization will result in poorer performance than eligibility trace-based optimization. Using the case of actor training for illustration, the term $\delta_{a,t}$, in that case, is an estimate of the true return derivative, as can be seen by Equation 3.34 where the critic's value gradient estimate λ is used to create $\delta_{a,t}$. For earlier time steps, such an estimate is likely to be less accurate than later time steps, as the critic slowly begins to be trained. Since momentum-based approaches accumulate the total weight update in the $\mathbf{M}$ term using $\delta_{a,t}$ from each time step, all subsequent actor updates will be influenced by these poorer prior $\delta_{a,t}$ estimates. Thus, intuitively, it can be said that momentum-based approaches introduce unwanted bias into the learning process. As opposed to eligibility trace approaches, where only the latest $\delta_{a,t}$ estimate is used.

A similar argument can be seen made in literature for the case of TD value function learning. For instance, Nichols [121], who applied both eligibility trace and momentum to the SARSA algorithm, empirically demonstrated that eligibility traces yielded an algorithm more likely to succeed and had greater sample efficiency. Similarly, Gupta [120] also showed that momentum on the TD algorithm hinders learning whereas eligibility traces on TD, i.e. TD(λ), results in better learning performance than the baseline TD algorithm.

3.4. IDHP Algorithm

The IDHP algorithm combines the various components introduced in Section 3.2: the RLS model, the actor, and the critic, to create the reinforcement learning agent. To initialize the algorithm, initial values for several algorithm variables as well as the algorithm's hyperparameters are set, Table 3.3 lists all the variables and hyperparameters to be initialized. The initialization values stated in Table 3.3 are either decided based on a combination of theory plus trial and error, the main heuristic used to decide on their values is to keep the algorithm simple, so using identity or null matrices and numbers as much as possible.

For the undecided hyperparameters, the experiments conducted involve running several different sets of learning and decay rates, thus values for these parameters will be stated in Section 3.5 instead. On the other hand, the hyperparameters γ, τ, ρ and variable initialization will be identical across all experiments, thus their values are given in Table 3.3. Regarding the choice of values for γ, τ, and ρ, these values were taken directly from previous research, specifically from the works of Heyer [4] and Teirlinck [110].

Note that the eligibility trace variable is initialized even if the algorithm is set not to use eligibility traces, this is because the eligibility trace update procedure is simply the general form of the network update procedure. Specifically, one obtains the original network weight update equations by setting the eligibility decay rate λ to 0.

After initialization, the main algorithm loop begins. First, one MDP step is taken, by sampling an action from the actor given the initial MDP state and performing that action in the environment. Second, the appropriate η is selected depending on the current error and warmup period, and the Σ is adapted depending on the warmup period and model error. Third, the critic, actor, and model are updated using the appropriate equations: the critic is updated using either a single or two-step TD error with the target critic weights moving slightly towards the critic weights, the actor is updated using the eligibility trace variables with λ being 0 if traces are not used and λ non-zero otherwise, finally the RLS model is updated using the latest model state and actor action to produce an estimate for the system model to use in the next time step. The algorithm then repeats this cycle, until the episode duration has reached the limit T. The algorithm is summarized in Algorithm 5.

Table 3.3: IDHP initialization variables and hyperparameters.

Hyperparameters		Variable initialization			
Actor learning rates:	$\eta_{a,h}$,$\eta_{a,l}$	Eligibility trace:	$\mathbf{E}_0$	$=$	$\mathbf{0}$
Critic learning rates:	$\eta_{c,h}$,$\eta_{c,l}$	RLS model parameter:	Θ_0	$=$	$\mathbf{0}$
Eligibility trace decay rates:	λ_h,λ_l	RLS model covariance:	Σ_0	$=$	$10^6 \cdot \mathbb{I}$
MDP reward scaling factor:	κ	actor-network weights:	$W_{a,0}$	$\sim$	$\mathcal{N}(0,\, 0.1^2)$
MDP reward discount factor:	$\gamma = 0.6$	critic-network weights:	$W_{c,0}$	$\sim$	$\mathcal{N}(0, 0.1^2)$
Target critic mixing factor:	$\tau = 0.01$	Model states:	x_0	$=$	$\mathbf{0}$
RLS forgetting factor:	$\rho = 1$	MDP states:	s_0	$=$	$\mathbf{0}$
		Model & MDP action:	a_0	$=$	$\mathbf{0}$

Algorithm 5 IDHP algorithm.

1: **Initialize**:
Set initial variable values and hyperparameters listed in Table 3.3.
2: **Online loop**:
For $t = 1$ **to** T/dt **do**
// *Sample and perform action*
$a_t \leftarrow \pi(s_t; W_{a,t})$
$s_{t+1}, r_{t+1} \leftarrow$ Environment(a_t)
// *Adapt* η *and* Σ *accordingly*

$$\eta_a, \eta_c \leftarrow \begin{cases} \eta_{\cdot,h} & \text{if} \quad t < 3 \text{ or } ||e_{[t-2,t]}|| > 1 \\ \eta_{\cdot,l} & \text{otherwise} \end{cases}$$

$$\Sigma_{t+1} \leftarrow \begin{cases} \text{Reset to } \Sigma_0 & \text{if} \quad ||\epsilon_t|| > 9 \times 10^{-5} \\ \text{Update using Equation 3.14} & \text{otherwise} \end{cases}$$

$$\Theta_{t+1} \leftarrow \begin{cases} \text{Reset to } \Theta_0 & \text{if} \quad ||\epsilon_t|| > 9 \times 10^{-5} \\ \text{Update using Equation 3.13} & \text{otherwise} \end{cases}$$

// *Update critic and target critic*
If using multi-step IDHP **then**

$$\delta_t = \lambda_{t-2} - \left(\frac{\partial r_{t-2}}{\partial x_{t-2}} + \gamma \frac{\partial r_{t-1}}{\partial x_{t-1}} \frac{\partial x_{t-1}}{\partial x_{t-2}}\bigg|_{t-1} + \gamma^2 \lambda'_t \frac{\partial x_t}{\partial x_{t-1}}\bigg|_t \frac{\partial x_{t-1}}{\partial x_{t-2}}\bigg|_{t-1}\right)$$

Else

$$\delta_t = \lambda_{t-1} - \frac{\partial r_{t-1}}{\partial x_{t-1}} - \gamma \lambda'_t \frac{\partial x_t}{\partial x_{t-1}}\bigg|_t$$

End If

$$\frac{\partial E_t}{\partial W_c} = \delta_t \frac{\partial \lambda(\cdot)}{\partial W_c}$$
$$W_{c,t+1} = W_{c,t} + \eta_c \frac{\partial E_t}{\partial W_c}$$
$$W_{c',t} = \tau W_{c,t-1} + (1-\tau) W_{c',t-1}$$

// *Update actor*

$\mathbf{E}_t \leftarrow$ accumulating trace (3.32) or replacing trace (3.33) ▷ $\lambda = 0$ *if not using eligibility traces*

$$\frac{\partial R_t}{\partial W_a} = \left[\frac{\partial r_{t-1}}{\partial x_t} + \gamma \lambda(s_t)\right] G_t \mathbf{E}_t$$
$$W_{a,t+1} = W_{a,t} + \eta_a \frac{\partial R_t}{\partial W_a}$$

// *Update model*
$F_{t+1}, G_{t+1} \leftarrow$Algorithm 4

3.5. Experiments

The experiments conducted will evaluate and compare the proposed augmentations against the baseline IDHP algorithm. While two augmentations were proposed, three augmented IDHP algorithms can be created:

1. IDHP(λ): baseline IDHP with the actor augmented by accumulating eligibility trace.
2. MIDHP: baseline IDHP with the critic augmented by multi-step update.
3. MIDHP(λ): baseline IDHP with the critic augmented by multi-step update and the actor augmented by replacing eligibility trace.

First, one study into how the augmentations affect the network gradients in the IDHP algorithm is conducted. This study will present a high-level discussion on the observed differences between the four algorithms when it comes to actor and critic updates over time. This study will run each algorithm on a sinusoidal AoA tracking problem for 5 s. The hyperparameters for this study were chosen out of convenience and kept simple, as the main focus here is on the augmentations. Meaning the most important factor when it comes to the hyperparameters used is for all algorithms to use the same values. A second consideration when choosing the hyperparameters was in concern of elucidating the differences between all algorithms, in order for the results to be more readily understandable. To this end, the eligibility decay rate λ was chosen to be at a relatively high value of 0.9, such that the results of this study will more clearly show the effect of using eligibility traces. The hyperparameters used for this study are reported in Table 3.4.

Table 3.4: Hyperparameters used during the weight gradients study.

Hyperparameters		
κ	$=$	1000
$\eta_{a,h}, \eta_{a,l}$	$=$	$2.0, 0.02$
$\eta_{c,h}, \eta_{c,l}$	$=$	$0.2, 0$
$\lambda_{a,h}, \lambda_{a,l}$	$=$	$0.9, 0.9$

Then, two Mone Carlo experiments will be conducted to study the effect of the augmentations on IDHP in terms of how they control the aircraft.

The first of such experiments involves studying the effect of only incorporating IDHP with each of the augmentations, where no changes to the hyperparameters or random number seeds used will be made. Since the initial network weights are sampled from a probability distribution, fixing the set of random number seeds used is important, as this ensures that the only differences across the four algorithms are the augmentations present, or lack thereof. All four agents will be given a step reference signal to track with **no faults** being introduced, 100 flights per algorithm will be conducted with the agent at the beginning of each flight being initialized to the initial variables listed under Table 3.3. Here, the main metrics for comparison are settling time t_s, defined as the time for tracking error to settle below 1.5 deg, and final error e_f, which is the absolute tracking error at the end of an episode/flight. The first metric, t_s, gives explicit information on how long the tracking error takes to settle, this is considered to be an indirect way of looking at how long the agents take to converge to their final policy; if an agent takes more time to converge to their final policy, it should take more time for the tracking error to settle, and vice versa. The second metric, e_f, is used to indicate the quality of the converged policy; a policy more optimal than another should logically have a smaller tracking error at the end of the episode. This experiment will use the same hyperparameters as the weight gradient study, except for the trace decay rates λ, where less aggressive values are used for better learning stability.

The second experiment involves studying the impact of the proposed augmentations on IDHP, by incorporating IDHP with each augmentation and tuning the hyperparameters of each algorithm. This means that different hyperparameters are used for each algorithm. In this experiment, the agents will be given a sinusoidal reference signal to track, while being **introduced to faults** 20 s into the flight. Just like the first experiment, each agent will repeat this control problem 100 times. For hyperparameter tuning, the algorithm performances are optimized for the cg shift fault scenario to minimize tracking

error, this resulted in different hyperparameters for each studied algorithm. A random search algorithm was used in these hyperparameter tuning procedures, where 15 hyperparameters are tested in parallel to speed up optimization using the multi-processing library of Python. Once again, the set of random number seeds used in all four algorithms is fixed. Here, the main metric used is the transient absolute tracking error e from the fault start to 10 s after. This metric is used to give an indication of how quickly the agents can adapt to the change in the MDP, and thus how adaptive the flight controller can be to a fault, a more adaptive controller should logically have a smaller error in the transient period after a fault. Specifications of the two experiments along with the hyperparameters used are summarized in Table 3.5.

Table 3.5: Specifications of the two experiments, experiment 1 uses the same hyperparameters for all algorithms, while experiment 2 uses different hyperparameters for each algorithm.

Experiment 1		Experiment 2	
Reference signal [deg]:	$\alpha_{ref,t} = 10$	Reference signal [deg]:	$\alpha_{ref,t} = 5\sin(2\pi\frac{t}{10})$
Metrics:	t_s, e_f	Metric:	$e = \sum_{t=20s}^{30s} \|\|\alpha_t - \alpha_{ref,t}\|\|$
Faults:	*None*	Faults @ 20 s:	*Shifted CG, damped elevator, inverted elevator*
Hyperparameters:		Hyperparameters:	
IDHP, IDHP(λ), MIDHP, MIDHP(λ)		IDHP	IDHP(λ)
$\kappa = 1000$		$\kappa = 1320$	$\kappa = 1211$
$\eta_{a,h}, \eta_{a,l} = 2.0, 0.02$		$\eta_{a,h}, \eta_{a,l} = 4.273, 0.059$	$\eta_{a,h}, \eta_{a,l} = 3.568, 0.054$
$\eta_{c,h}, \eta_{c,l} = 0.2, 0$		$\eta_{c,h}, \eta_{c,l} = 0.471, 0$	$\eta_{c,h}, \eta_{c,l} = 0.472, 0$
$\lambda_{a,h}, \lambda_{a,l} = 0.5, 0.1$		$\lambda_{a,h}, \lambda_{a,l} = 0, 0$	$\lambda_{a,h}, \lambda_{a,l} = 0.57, 0.257$
		MIDHP	MIDHP(λ)
		$\kappa = 1254$	$\kappa = 1289$
		$\eta_{a,h}, \eta_{a,l} = 3.2, 0.058$	$\eta_{a,h}, \eta_{a,l} = 3.398, 0.09$
		$\eta_{c,h}, \eta_{c,l} = 0.462, 0$	$\eta_{c,h}, \eta_{c,l} = 0.378, 0$
		$\lambda_{a,h}, \lambda_{a,l} = 0, 0$	$\lambda_{a,h}, \lambda_{a,l} = 0.236, 0.137$

3.5.1. Reliability of Results: Coefficient of Variation

To demonstrate the reliability of the gathered metrics in the Monte Carlo experiments, the sample group's coefficient of variation C_v should stabilize. Broadly speaking, the more samples that can be drawn for a study, the more reliable the statistics are. This is because the underlying distribution that represents the data can be more accurately approximated, assuming that the distributions are stationary. To show that the number of samples gathered is sufficient to approximate their underlying distribution, one can calculate the ratio of standard deviation to mean: this is the C_v statistic:

$$C_{v_i} = \frac{\sigma_i}{\mu_i} \tag{3.36}$$

Where the i subscript denotes the number of samples used to determine the variable. I.e., $C_{v_{10}}$ is the standard deviation of 10 samples divided by the mean of the same 10 samples. One simple way to calculate this variable is to do so as the Monte Carlo study is conducted, for every new sample n, an associated C_{v_n} can be calculated using the n samples collected thus far. By continuously calculating this variable throughout a Monte Carlo study, one should expect to see the C_v value stabilize at some stable level, at which point it can be said that the Monte Carlo study has *converged* and that the resulting

statistics are reliable to some degree. A similar measure of reliability is used by Ballio and Guadagnini [129] who use the stabilization of mean and variance over the number of samples as a measure of convergence.

3.5.2. Statistical Significance of Results: t-test and a-test

To provide further rigour on the results gathered, the statistical significance of the observations can be found quantitatively. This is done in two ways, the first is using Student's t-test [130], and the second is using Vargha and Delaney's a-test[131].

Student's t-test is one form of statistical test used to conclude whether two groups of samples can be said to belong to the same population or underlying distribution. With the t-test, a null hypothesis is made stating that the mean and variance of the population underlying the two groups are the same. Thus, disproving this null hypothesis would mean the two groups in fact belong to different populations. To prove or disprove this null hypothesis, the t-test produces a p-value, which on a high level can be interpreted as the probability that the null hypothesis is true and ranges between 0 and 1. This p-value should be below a threshold in order to conclude that the null hypothesis is false, the common choice for such a threshold is 0.05 or 5%, which is the threshold chosen in the present research as well.

Vargha and Delaney's a-test is another form of statistical test, this test is used to provide information on the magnitude of difference between two groups of samples. In more mathematical terms, this statistical test determines what the stochastic ordering of the two groups is: whether one random variable is larger, smaller, or equal, to another random variable. In the a-test, an A-value is calculated between two groups of samples a and b representing the stochastic order of the two groups, this value ranges between 0 and 1. This A-value is the stochastic superiority of group a over group b, it $A > 0.5$, then group a generally has sample values which are higher than group b, i.e. group a is stochastically superior compared to group b. The reverse is also true, if $A < 0.5$, then group a is stochastically inferior compared to group b.

Applying the a-test to hypothetical settling time t_s samples as an example, suppose an a-test between t_s of MIDHP and IDHP returns an A-value of 0.3, this means that the settling times of MIDHP are generally smaller than the settling times of IDHP. Then, if an a-test between t_s of IDHP(λ) and IDHP returns an A-value of 0.55, two conclusions can now be drawn: first is that the settling times of IDHP(λ) are generally larger than IDHP, and second is that the settling times of IDHP(λ) are also larger than that of MIDHP. Final note, a property of the a-test is that the A-value is symmetric, i.e. if the A-value between group a and b is 0.3, then the A-value between group b and a is 0.7. This property follows from the definition of A from Vargha and Delanye's work [131].

By using these two statistical tests in conjunction, the metrics of different algorithms can be verified to be statistically significant as well as how they rank relative to each other, done using Student's t-test and Vargha Delaney's a-test respectively, foregoing subjective conclusions.

3.6. Results & Discussions

This section presents simulated flight control performances of the IDHP algorithm and its derivatives. This section presents the simulation results on the nominal performance and fault tolerance of the four present algorithms: IDHP, IDHP(λ), MIDHP, and MIDHP(λ). The main goal of this section is to provide a quantitative and qualitative evaluation regarding how the proposed augmentations affect the performance of the baseline IDHP algorithm through the results.

3.6.1. Weight Gradients Study

The four algorithms are given a test MDP problem to tackle to gather time traces of the algorithm's actor and critic updates. The chosen problem is a control task where the aircraft's AoA is controlled to follow a sinusidal AoA reference signal. The gradient time traces are shown in Figure 3.6, from the top row of figures to the bottom row, the order of gradient plots are: IDHP, IDHP(λ), MIDHP, and MIDHP(λ). The left column of plots correspond to the actor updates, while the right column correspond to the critic updates.

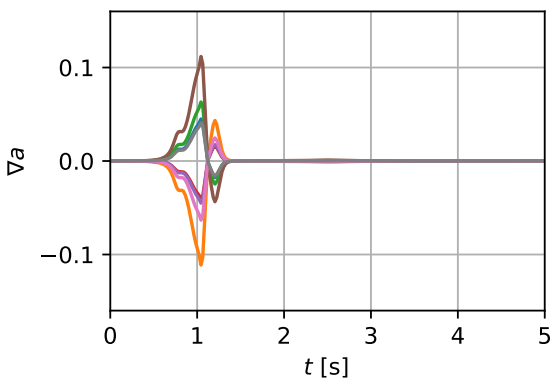
Comparing the second row to the first row, the actor update time traces can be seen to be shifted leftwards in the second row, same observation is made for the critic updates. Additionally, the magnitude

of actor updates is in general higher, even though the shape of the update time traces between the first and second row are very similar for both actor and critic updates. This makes sense on a theoretical level, the use of eligibility traces in IDHP(λ) on the actor means that the updates which the actor receives will include the updates of the previous time steps. Looking at the figures in row 1, the weight updates which the actor receives do not change in sign often, or exhibit any oscillatory action, for the first second the updates are either monotonically increasing or decreasing. Therefore, when these updates are accumulated using eligibility traces, it can be predicted that the actor updates will simply be a quicker and higher magnitude version of the updates from the baseline algorithm.

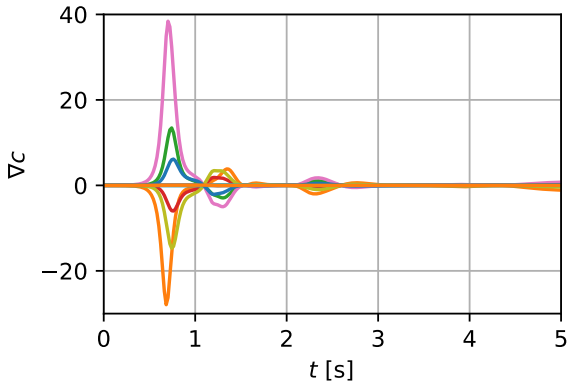
Comparing the third row of figures to the first, this time the actor updates are more similar than in the first comparison between the first and second row. However, the critic updates between the first and the third row look dissimilar where in the third row a lot more new updates, albeit relatively small magnitudes, are happening after the initial spike of critic updates. This shows that using a multi-step TD error for updating the critic introduced new update steps to the critic which would not have been present otherwise. In the upcoming Monte Carlo experiments, it will be seen what the effect of this is on the algorithm's control performance.

Finally, comparing the fourth row of figures to the first row. One observation here is that the differences between the critic update of the fourth row to the first row are very similar to that of the third row and the first row: smaller updates are made to the critic after the initial spike of large updates. To add further, it would appear that the use of replacing eligibility traces on the actor updates was relatively mild compared to the effect which accumulating eligibility traces had. In fact, the only difference that can be discerned between the actor updates of the fourth row and that of the first row is the oscillatory updates around 1.3 s, seen after the initial update spike of updates. Otherwise, the differences in actor updates between the two rows are minute.

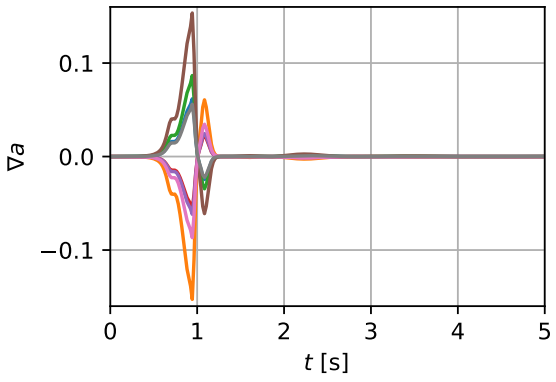
Despite some differences existing on close inspection, the four rows of figures seem similar at a glance, which might lead one to conclude that the augmentations did not meaningfully affect the algorithm. But to really understand how the augmentations affected the characteristics of the algorithm which are important, analysis of the Monte Carlo experiments results are needed.



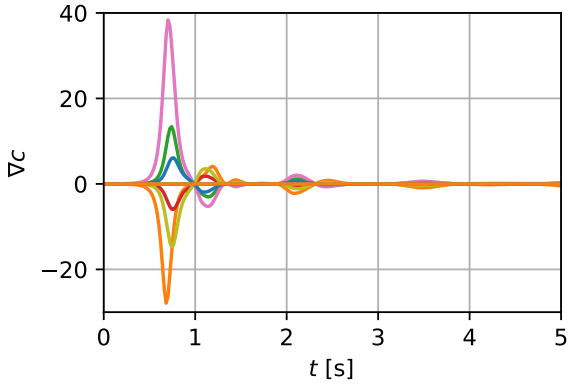
(a) IDHP actor updates.



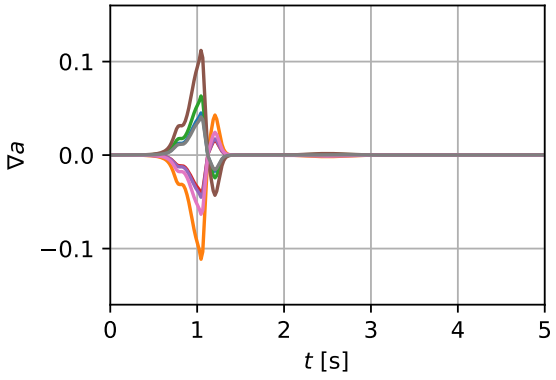
(b) IDHP critc updates.



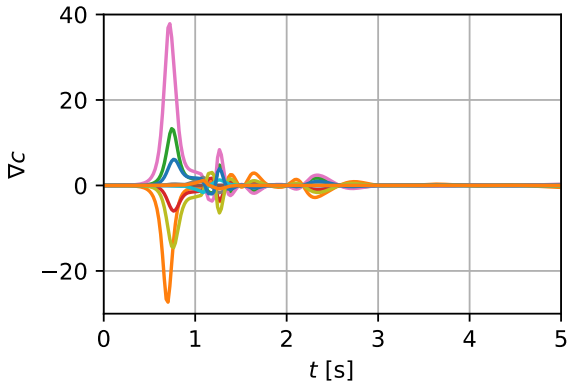
(c) IDHP(λ) actor updates.



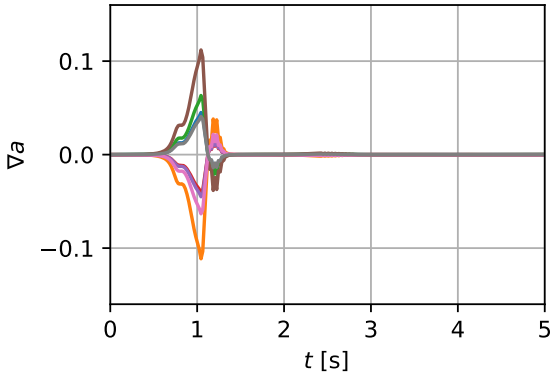
(d) IDHP(λ) critic updates.



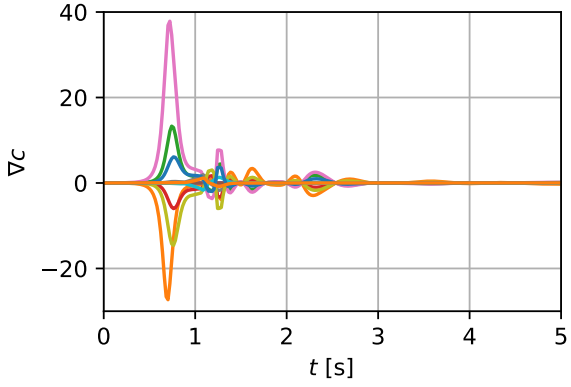
(e) MIDHP actor updates.



(f) MIDHP critic updates.



(g) MIDHP(λ) actor updates.



(h) MIDHP(λ) critic updates.

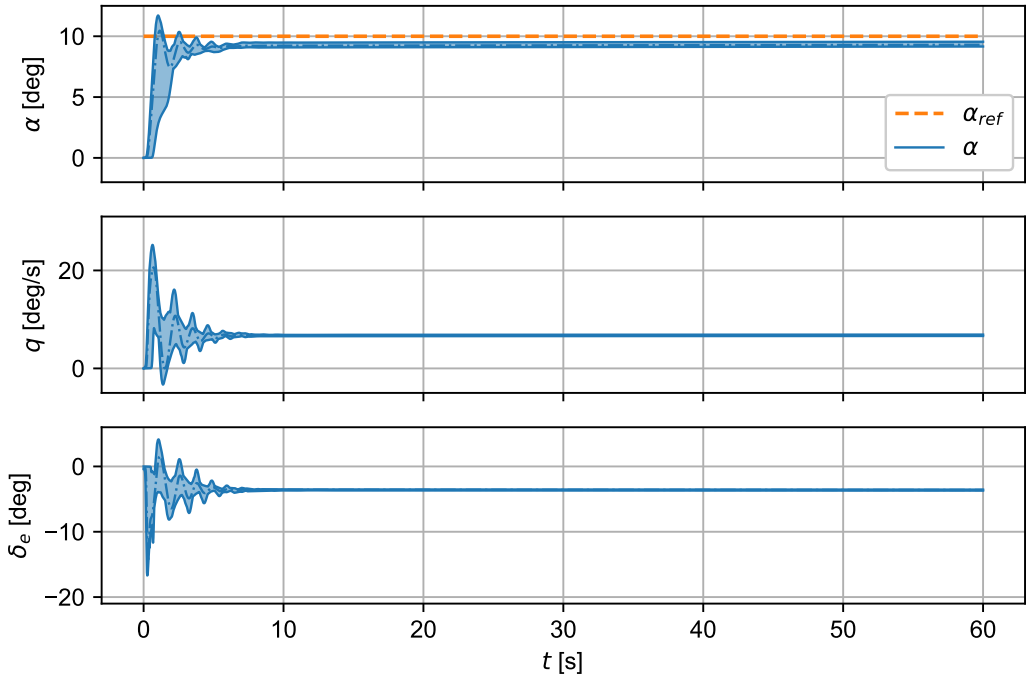
Figure 3.6: Actor and critic update over 5 s of the four algorithms solving the same control task.

3.6.2. Experiment 1

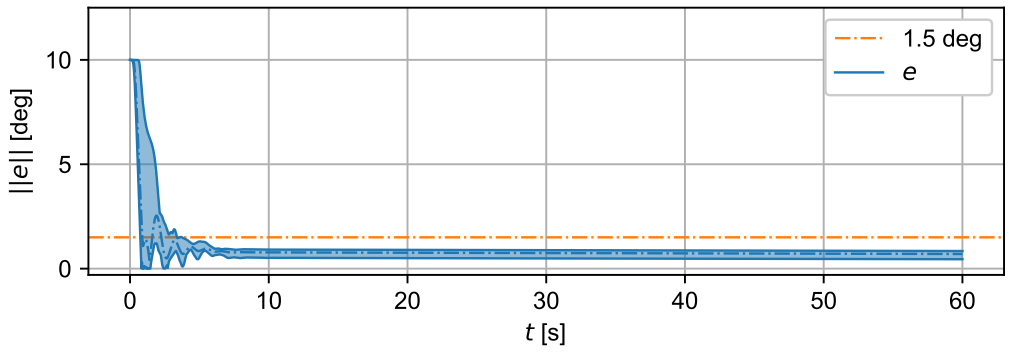
The control task for this experiment is a step AoA reference tracking problem, roughly speaking, the optimal control action is for the controller to smoothly increase the deflection of the elevator until the aircraft's AoA meets the reference value. Shown in Figure 3.7 is the Monte Carlo response of the IDHP agent for experiment 1, with the mean responses and their spread plotted over time.

From Figure 3.7a, the response akin to that of a purely proportional feedback controller can be seen, where the system state rises sharply and oscillates before settling around the reference value. This observation of a proportional controller-like response is expected, because the policy input, the MDP state space, is simply the tracking error. This means that the output of the policy function can only be some factor multiplied by the tracking error, thus the policy function behaves like a proportional gain.

The tracking error of this controller can be seen in Figure 3.7b, showing that the tracking error settles below 1.5 deg around 2 to 3 s and has a non-zero error at the end of each episode.



(a) State and action time traces.

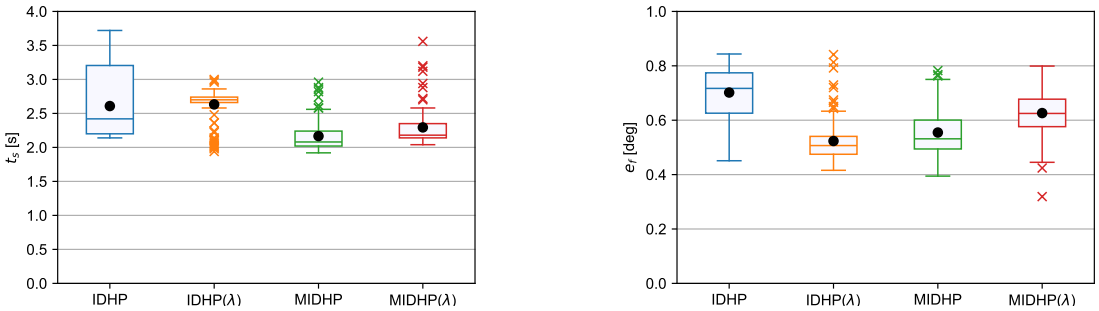


(b) AoA tracking error time trace with the 1.5 deg settling time threshold shown in dash-dotted orange, minimum and maximum shown by shaded area, mean shown by dash dot line.

Figure 3.7: IDHP step tracking result.

To compare the performances of all four algorithms, their corresponding t_s and e_f metrics from the 100

Monte Carlo runs are presented as boxplot diagrams in Figure 3.8. The key numerical statistics from the boxplots are presented in Table 3.6.



(a) Settling time t_s. **(b)** Final error e_f.

Figure 3.8: Boxplots of experiment 1 metrics from the 100 runs conducted of each algorithm, black dots are the mean values.

Table 3.6: Settling time t_f and final error e_f statistics.

	t_f [s]				e_f [deg]			
	IDHP	IDHP(λ)	MIDHP	MIDHP(λ)	IDHP	IDHP(λ)	MIDHP	MIDHP(λ)
Maximum:	3.72	3	2.96	3.56	0.84	0.84	0.78	0.8
Minimum:	2.14	1.94	1.92	2.04	0.44	0.42	0.38	0.32
Upper quartile:	3.2	2.73	2.24	2.34	0.78	0.54	0.6	0.68
Lower quartile:	2.2	2.66	2.02	2.14	0.62	0.48	0.49	0.57
Median:	2.42	2.7	2.08	2.18	0.72	0.51	0.53	0.62
Mean:	2.6	2.63	2.17	2.29	0.7	0.52	0.55	0.53
No. outliers (above, below):	$\{0, 0\}$	$\{3, 14\}$	$\{7, 0\}$	$\{8, 0\}$	$\{0, 0\}$	$\{9, 0\}$	$\{3, 0\}$	$\{0, 2\}$

Starting with the analysis of the t_s comparisons shown in Figure 3.8a, the range of settling times for all algorithms ranged between 1.9 to 3.75 s, where IDHP and MIDHP produced the highest and lowest settling time respectively. The baseline IDHP algorithm overall had the widest spread in settling time with an Inter Quartile Range (IQR) of 1 s, this is in contrast to the derivative algorithms which all had a much smaller IQR, which at the smallest was 0.7 for IDHP(λ) and at the largest was 0.22 for MIDHP. While the spread of the median 50% of the settling times was much tighter for the derivative algorithms, their median times were generally lower. For IDHP, the median settling time was 2.42 s, while MIDHP had a median time as low as 2.08 s, which is 14% quicker. IDHP(λ), however, had a higher median settling time, at 2.7 s or 11.6% higher. Despite this higher median settling time, IDHP(λ) yielded a lower minimum and maximum settling time compared to IDHP, the same can be said when comparing MIDHP or MIDHP(λ) to IDHP where IDHP had the higher settling time in all the quantiles.

Thus overall, it can be said that two of the three derivative algorithms improved the settling time of the flight controller, with the exception of IDHP(λ) which arguably increased the settling time slightly. Furthermore, the augmentations in general result in less sensitivity to network initialization, evident by the much smaller settling time IQR, which is a desirable quality in controller robustness. That being said, any practical controller should not be initialized by random sampling from any distribution, but instead use some pre-determined values, which could be decided on randomly or through more systematic means.

Moving on to the analysis of e_f, the range of e_f stood between 0.84 and 0.32 deg, where IDHP and MIDHP(λ) produced controllers that gave the highest and lowest e_f respectively. In this metric, the IQR of all four algorithms are more similar than with the t_s metric but still significant, with the highest being

0.16 deg from IDHP and the lowest being 0.06 deg from IDHP(λ), which is a 62.5% difference. Another distinction between IDHP and its derivatives can be gleaned from these statistics. The median and mean e_f of the derivative algorithms are all smaller than IDHP, the smallest improvement is found in MIDHP(λ) which has a 13.9% improvement, while the largest improvement is found in IDHP(λ) which had an e_f that was 29.2% smaller at 0.51 deg.

Considering the whole set of data, a clear picture can be seen by observing Figure 3.8b that IDHP generally has a worse e_f. This is to say that the augmentations allowed the agent to learn a policy which reached a lower asymptotic error than the baseline algorithm under a step reference tracking task, albeit still with non-zero error.

With these statistics, it can be stated that the augmentations can improve the flight control performance of the IDHP algorithm since all factors aside from the augmentations have been held constant across all four Monte Carlo studies. A more nuanced point to be made is to what degree they improved performance. Of all the algorithms, MIDHP yielded the largest improvements in both t_s and e_f, for which the upper quartiles were both in the neighbourhood or lower than the lower quartile of the same statistics in the IDHP algorithm. This is to say that the multi-step augmentation provided the biggest likelihood for improvement in the controller's step tracking performance, as well as being more unilaterally better in both metrics. As opposed to IDHP(λ), which while having much lower e_f, was not likely to have lower t_s than the IDHP controller.

Coefficient of Variation

The C_v plots for t_s and e_f of the four algorithms are presented in Figure 3.9 and Figure 3.10, the ends of each plot have boxes bounding the graphs above and below to within 5% of the final C_v values. It can be seen that the C_v graphs mostly converge within this bounding box in the final few samples, meaning that the t_s and e_f metrics from the samples taken have mostly converged.

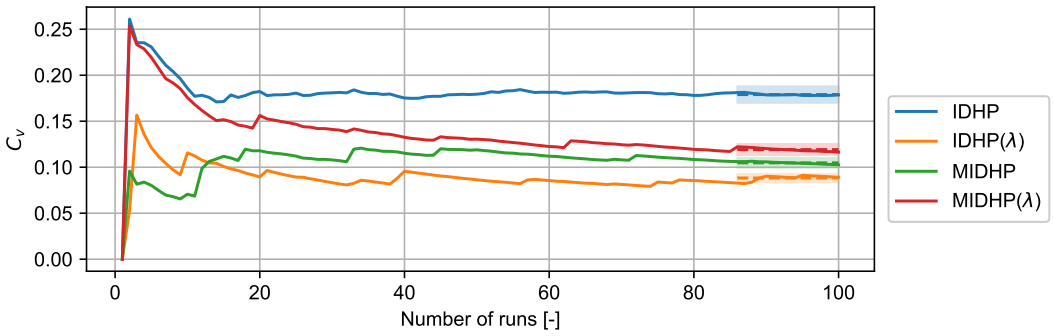


Figure 3.9: C_v plot for t_s, bounding boxes show mean of final 15 runs plus minus 5%.

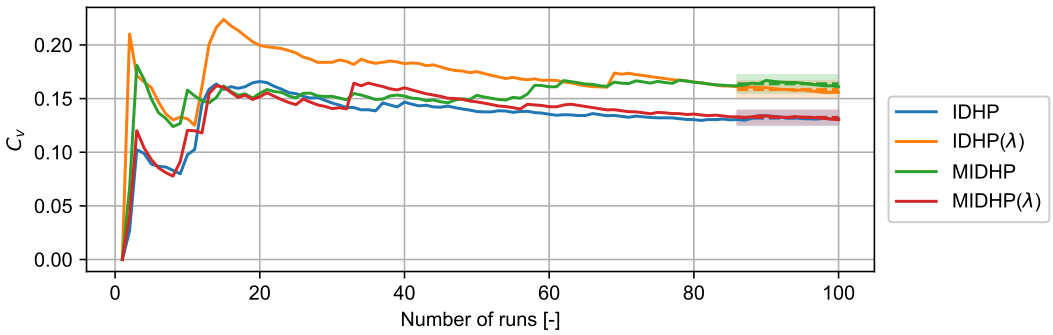


Figure 3.10: C_v plot for e_f, bounding boxes show mean of final 15 runs plus minus 5%.

Statistical Tests

A pair-wise statistical testing is done between the sampled metrics from each of the three derivative algorithms and the baseline IDHP algorithm. The statistical test results are summarized in Table 3.7. From these tests, it can be concluded that the difference between the derivative and baseline IDHP are all statistically significant except for the t_s samples of IDHP(λ), whose p-value is greater than 0.05. From these statistical tests, it is concluded that MIDHP showed the biggest improvement in nominal flight since it had the smallest A-value for t_s and the second smallest A-value for e_f. Which respectively indicates having the lowest t_s and second lowest e_f overall.

Table 3.7: Statistical testing results of experiment 1, p-values indicating statistically significant and insignificant differences shown in green and red respectively.

		Statistical tests		
		IDHP(λ) vs IDHP	MIDHP vs IDHP	MIDHP(λ) vs IDHP
t_s	p-value	0.6447	3.9650×10^{-15}	2.3549×10^{-8}
	A-value	0.573	0.1436	0.2653
e_f	p-value	1.0678×10^{-32}	1.7971×10^{-23}	5.3592×10^{-9}
	A-value	0.0862	0.1364	0.2683

3.6.3. Experiment 2

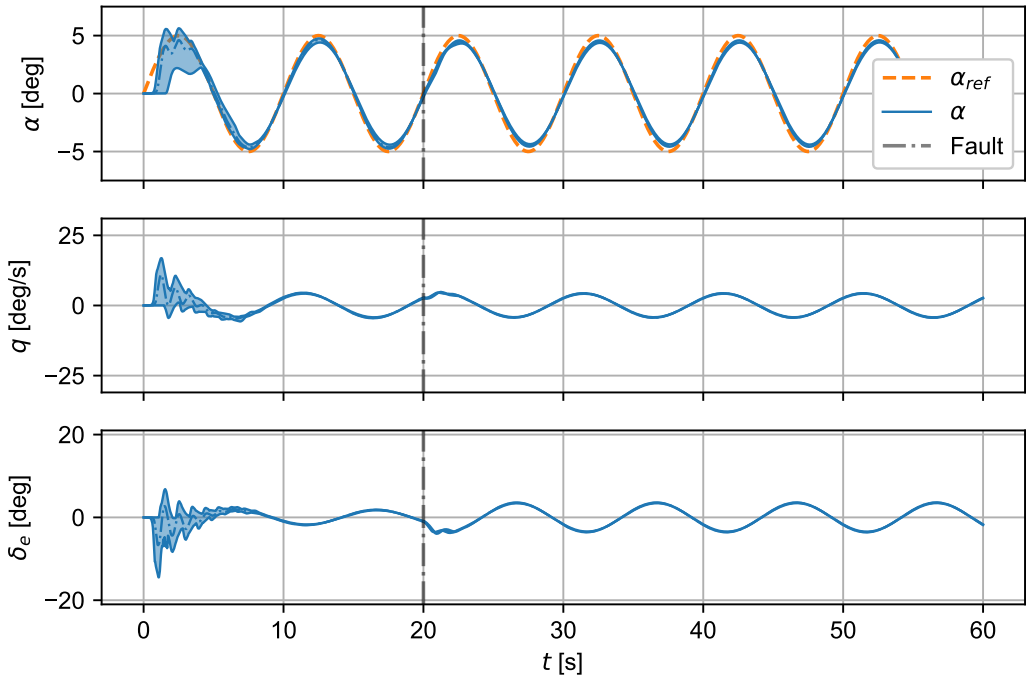
The control task for this experiment is a sinusoidal AoA reference tracking task with faults being introduced 20 s into the flight. The three faults of shifted CG, damped elevator, and inverted elevator as listed in Table 3.5 and explained in Section 3.1.3 are used. Since the introduction of augmentations changed what hyperparameters gave the optimal performance, it was decided that a comparison of the algorithms under tailored hyperparameters would give a broader and more fair evaluation of the algorithms.

This experiment's objective is to study the effect of the augmentations on the fault tolerance of the IDHP controller, by measuring the impact of introducing fault on the total tracking performance metric e of the aircraft between when the fault is introduced and 10 s after. This metric is effectively equivalent to measuring the settling time in a step tracking task, as a low e would mean that the controller steered the aircraft AoA quickly back to the reference value.

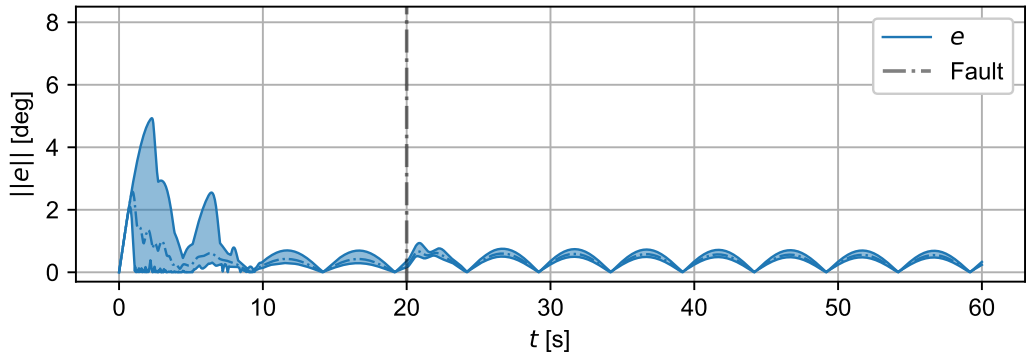
Figure 3.11 shows the tracking performance of the IDHP controller for the sinusoidal tracking task with the elevator damaged 20 s into the flight. As can be seen from Figure 3.11b, the introduction of the fault led to a deterioration in the controller's tracking performance, however, this did not last as the tracking error began to steadily decrease after the error's peak. In fact, the 100 different agents seemingly converged to a more similar policy than before the fault, as the spread of tracking errors in Figure 3.11b became tighter after the 20 s mark.

The IDHP agent continually identifies a system model using RLS, and it is interesting to observe the prediction error of this model. This can be done by inspecting the $||\epsilon||$ over time, where ϵ is the model innovation or prediction error. Such a plot is shown in Figure 3.12. Observing this graph, $||\epsilon||$ immediately before the fault has been steadily decreasing, showing that the model's prediction accuracy was gradually improving. Immediately after the fault, $||\epsilon||$ jumps up sharply past the RLS covariance reset threshold of 9×10^{-5}. This triggers the model to reset, and the RLS algorithm re-identifies the system dynamics. Thereafter, the model innovation can be observed to continuously fall, indicating the identified model dynamics are once again converging towards the true system dynamics.

To dive into the comparative performances of the four algorithms, the transient tracking error e from the Monte Carlo studies are once again graphed in boxplots, these graphs are presented and analyzed in the following texts.



(a) State and action time traces.



(b) AoA tracking error time trace.

Figure 3.11: IDHP sinusoidal tracking results with the elevator damage initiated at 20 s, minimum and maximum shown by the shaded area, mean shown by dash dot line.

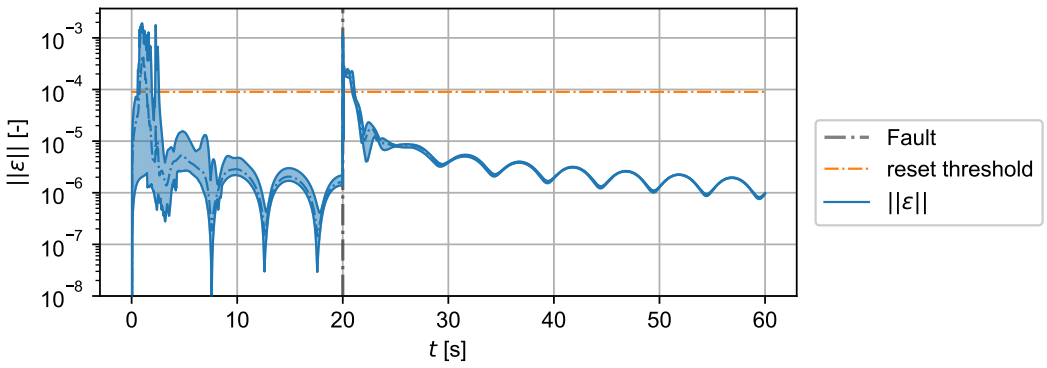


Figure 3.12: Monte Carlo result of RLS innovation norm $||\epsilon||$ over time, minimum and maximum shown by shaded area, mean shown by dash dot line.

Shifted Centre of Gravity

Figure 3.13 shows the e boxplots of the four algorithms after a shift in the CG, the numerical statistics are presented in Table 3.8. The values of e found with this fault ranged from a maximum of 4.49 deg from the IDHP(λ) algorithm to a minimum of 3.15 deg also from the same algorithm. The transient error of the IDHP(λ) controller is relatively large, as shown in Figure 3.13. This is in contrast with the MIDHP(λ) algorithm, which had a maximum and minimum of 4.2 and 3.41 deg, which is a 41% smaller spread. This difference can also be seen in the e time traces, which are shown zoomed in around the 20 s mark in Figure 3.14a. As a reference, the baseline result of IDHP is shown in Figure 3.14b.

While IDHP(λ) had the largest spread of e, it also had a lower median and lower quartile e than IDHP, meaning that IDHP(λ) had more runs whose error converged faster after the CG shift fault. The multi-step augmentations gave both a lower and higher overall e than the baseline IDHP e spread, where the spread of e from MIDHP(λ) was around the lower quartile of e samples from IDHP, therefore most MIDHP(λ) controllers recovered quicker from the CG shift fault. On the other hand, the group of e samples from MIDHP was slightly higher than the median e of IDHP.

The maximum difference between the median e of IDHP and any of its derivative algorithms is 5.77% between IDHP and MIDHP(λ), which had a median e of 3.81 and 3.89 deg respectively. This is a comparatively small improvement compared to the ones observed in experiment 1 under Section 3.6.2.

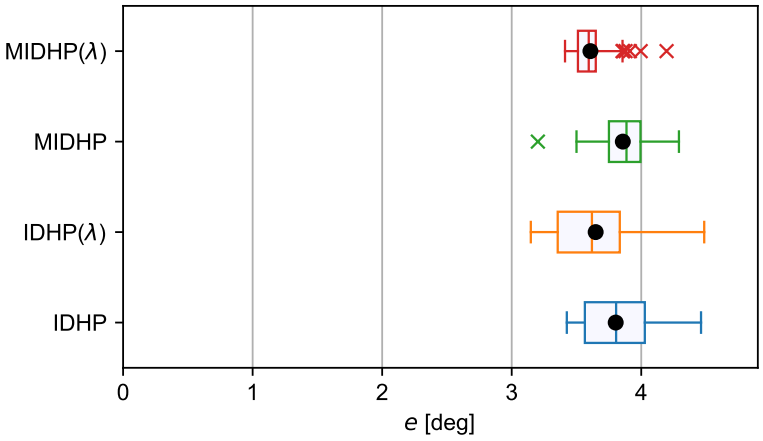
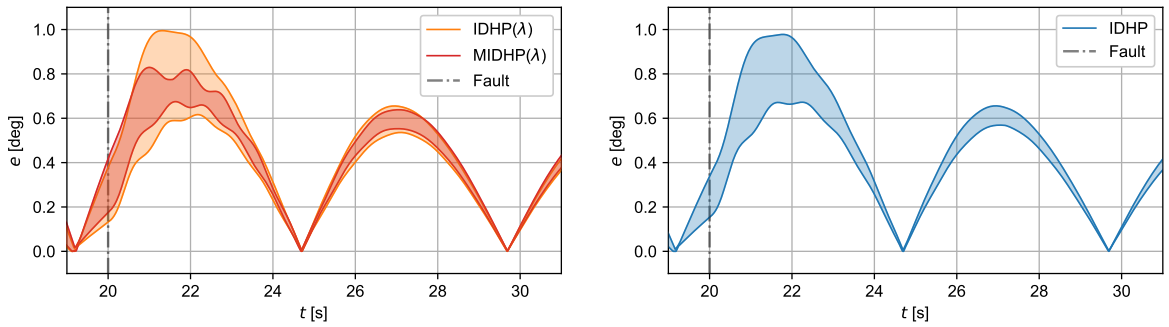


Figure 3.13: e boxplots of experiment 2 with the CG shift fault, black dots are the mean values.

Table 3.8: Transient error e statistics after the CG shift fault.

	e [deg]			
	IDHP	IDHP(λ)	MIDHP	MIDHP(λ)
Maximum:	4.46	4.49	4.29	4.2
Minimum:	3.43	3.15	3.2	3.41
Upper quartile:	4.02	3.83	3.99	3.65
Lower quartile:	3.56	3.36	3.75	3.51
Median:	3.81	3.62	3.89	3.59
Mean:	3.8	3.65	3.86	3.61
No. outliers (above, below):	$\{0, 0\}$	$\{0, 0\}$	$\{0, 1\}$	$\{6, 0\}$



(a) IDHP(λ) and MIDHP(λ) e.

(b) IDHP e.

Figure 3.14: Minimum and maximum$e$ error over time for three of the tested controllers.

The C_v plots for the group of e samples for the CG shift fault experiment is shown in Figure 3.15, where it can be seen the C_v stabilizes towards the end of the Monte Carlo runs.

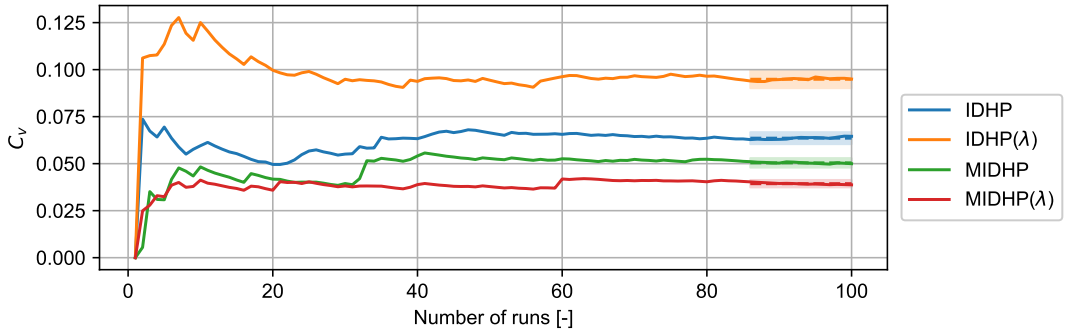


Figure 3.15: C_v plot for e under the CG shift fault in experiment 2, bounding boxes show mean of final 15 runs plus minus 5%.

The statistical testing results for the samples from this experiment are shown in Table 3.9. The statistical tests show that the difference between the derivative algorithms and the baseline are statistically significant, except for MIDHP versus IDHP where the p-value is 0.0813. It is also shown that MIDHP(λ) yielded the largest improvement in transient error since it's A-value is the smallest.

Table 3.9: Statistical testing results of experiment 2 with the shifted CG fault, p-values indicating statistically significant and insignificant differences shown in green and red respectively.

Statistical tests			
	IDHP(λ) vs IDHP	MIDHP vs IDHP	MIDHP(λ) vs IDHP
p-value	3.687×10^{-4}	0.0813	7.9148×10^{-11}
A-value	0.334	0.5762	0.2734

Damaged Elevator

The next fault scenario lowers the control effectiveness of the elevator by 50% to represent a damaged elevator. The group of e samples are shown in boxplots in Figure 3.16, and the statistics are numerically recorded in Table 3.10. At a quick glance, the boxplots for this fault case and the shifted CG scenario are very similar, see Figure 3.13: The IDHP(λ) algorithm has the largest spread, followed by MIDHP, then IDHP, and ending with MIDHP(λ) which has the smallest spread. Additionally, the median e value of MIDHP(λ) as well as its median 50% of the e values are smaller than that of the IDHP algorithm. This time, the median e of MIDHP(λ) is 3.14 deg which is 5.14% smaller than the median of IDHP which is

at 3.31 deg. Reading the statistics, MIDHP(λ) has a much more consistent spread of transient errors at a smaller spread than the baseline IDHP, meaning that it adapts better to the presented fault, this difference is also shown by the e time traces shown in Figure 3.17. Note that for the MIDHP(λ) time trace, the maximum e value was an outlier that is far from the upper quartile range, thus most of the controllers have a smaller e than is apparent in Figure 3.17.

In this fault scenario, the combination of multi-step TD update on the critic and eligibility traces for actor update contributed to a more robust and consistent controller: the recovery from a damaged elevator was quicker and the agent converged to a more optimal policy than the baseline IDHP.

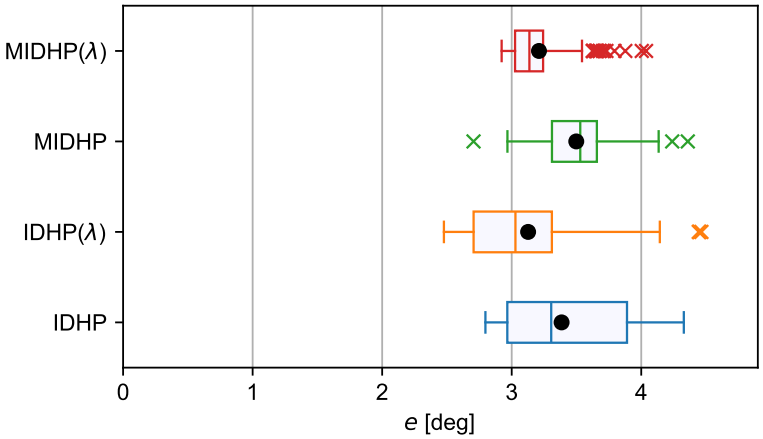


Figure 3.16: e boxplots of experiment 2 with the damaged elevator fault of each algorithm, black dots are the mean values.

Table 3.10: Transient error e statistics after the damaged elevator fault.

	e [deg]			
	IDHP	IDHP(λ)	MIDHP	MIDHP(λ)
Maximum:	4.33	4.46	4.36	4.04
Minimum:	3.80	2.48	2.70	2.92
Upper quartile:	3.89	3.31	3.66	3.24
Lower quartile:	2.97	2.71	3.31	3.03
Median:	3.31	3.01	3.53	3.14
Mean:	3.39	3.13	3.50	3.21
No. outliers (above, below):	$\{0, 0\}$	$\{4, 0\}$	$\{2, 1\}$	$\{16, 0\}$

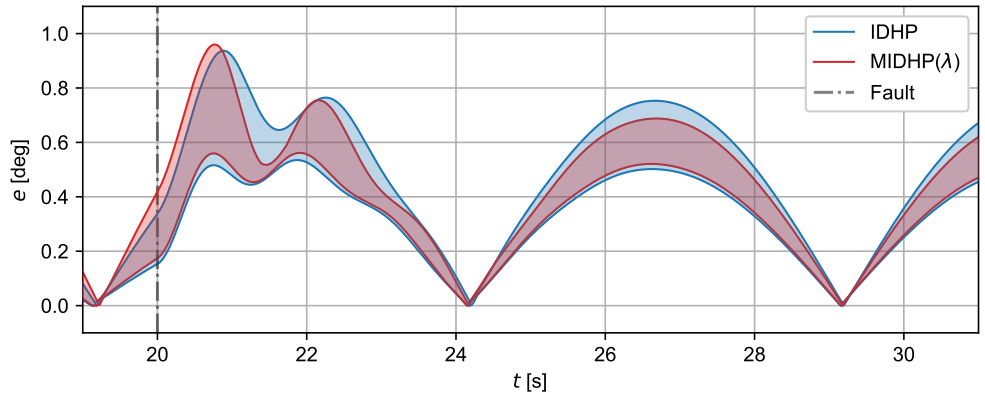


Figure 3.17: Experiment 2 with damaged elevator, minimum and maximum e over time for IDHP and MIDHP(λ).

The C_v plots of the e sample population for the damaged elevator fault experiment are shown in Figure 3.15, where it can be seen that once again the C_v stabilizes towards the end of the Monte Carlo runs.

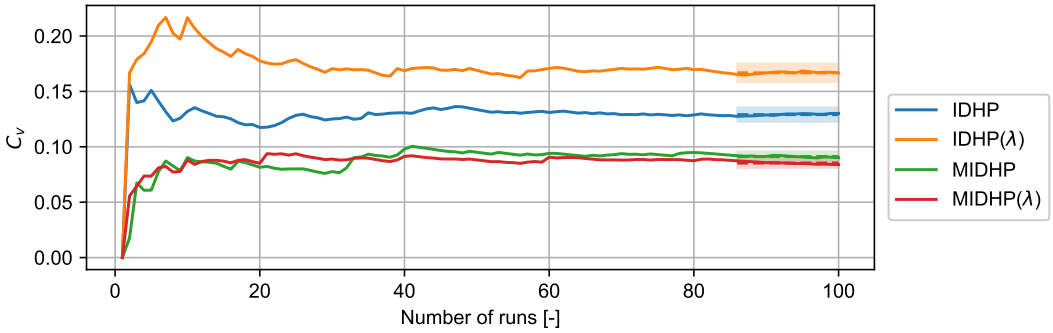


Figure 3.18: C_v plot for e under the damaged elevator fault in experiment 2, bounding boxes show mean of final 15 runs plus minus 5%.

The statistical testing results for the samples from this experiment are shown in Table 3.11. The statistical tests show that the difference of all three derivative algorithms from the baseline is statistically significant and that IDHP(λ) yielded the biggest improvement, with the lowest A-value.

Table 3.11: Statistical testing results of experiment 2 with the damaged elevator fault, p-values indicating statistically significant and insignificant differences shown in green and red respectively.

Statistical tests			
	IDHP(λ) vs IDHP	MIDHP vs IDHP	MIDHP(λ) vs IDHP
p-value	2.1594×10^{-4}	0.0388	9.2228×10^{-4}
A-value	0.3145	0.6037	0.4069

Reversed Elevator

The last fault scenario is the reversed elevator fault. The boxplot of transient errors e from stable controllers is shown in Figure 3.19, and the associated statistics are recorded in Table 3.12, a stable controller here means one which does not produce excessively oscillatory or unresponsive actions. Note that the final row of Table 3.12 records the number of agents which had an unstable controller after the reversed elevator fault is triggered. Furthermore, only the e of stable controllers are recorded

and plotted in the boxplots such that these diagrams illustrate the performance of the algorithms only when they converge to a stable controller; leaving the matter of likelihood for the algorithm to produce stable controllers to be seen through the number of diverged controllers.

Reading from Table 3.12, it can be seen that for IDHP(λ), 98% of controllers were unstable. For other algorithms, while the portion of stable controllers is higher, the portion of unstable controllers is still significantly high. The most successful algorithm in this fault scenario was IDHP which had 74 stable controllers, followed closely by MIDHP(λ) which had 71 stable controllers.

From Figure 3.19, it can be seen that the stable IDHP controllers generally had the lowest error from the four algorithm's controllers. The median error of stable IDHP controllers is 12.02 deg, and the next closest median e is MIDHP(λ) whose median e is 13.32 deg or 9.76% higher.

Overall, the baseline IDHP algorithm can handle the reversed elevator fault scenario better than any of its derivative algorithms, both in terms of proportion and quality of stable controllers.

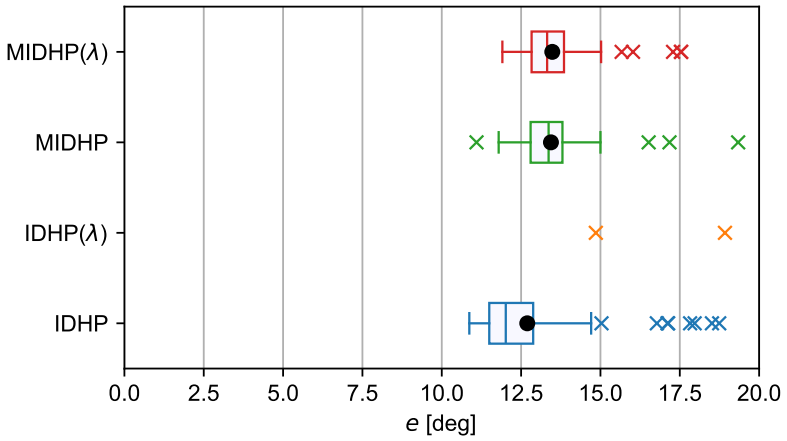


Figure 3.19: e boxplots of experiment 2 with the reversed elevator fault of each algorithm, black dots are the mean values.

Table 3.12: Transient error e statistics after the reversed elevator fault.

	e [deg]			
	IDHP	IDHP(λ)	MIDHP	MIDHP(λ)
Maximum:	18.74	18.9	19.33	17.53
Minimum:	10.86	14.82	11.08	11.90
Upper quartile:	12.86	3.31	13.82	13.84
Lower quartile:	11.50	2.71	12.78	12.82
Median:	12.02	3.01	13.34	13.32
Mean:	12.70	3.13	13.42	13.49
No. outliers (above, below):	$\{8, 0\}$	$\{0, 0\}$	$\{2, 1\}$	$\{5, 0\}$
No. diverged :	26	98	52	29

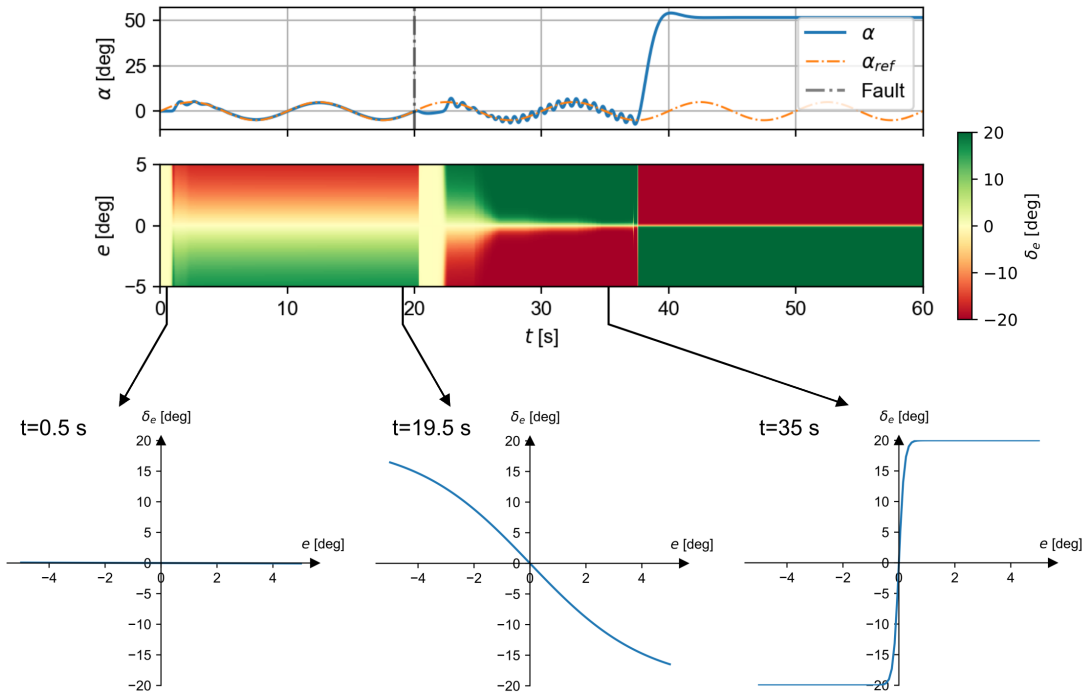
This fault was the most difficult scenario for all algorithms to adapt to, where many of the Monte Carlo runs had agents which would converge to an unstable controller, exhibiting highly oscillatory control action or stuck at full elevator deflection after the fault was introduced.

To investigate this matter, it would be useful to understand what the policy function looks like for a controller which is stable versus one which is unstable. Such a study is simple to conduct if only one instantaneous policy is studied, or if the policy function does not adapt and change over time. However, the policy function in the IDHP algorithm does evolve over time, with the policy function producing a different mapping at every time step of the simulation. Therefore, to be able to represent this temporal

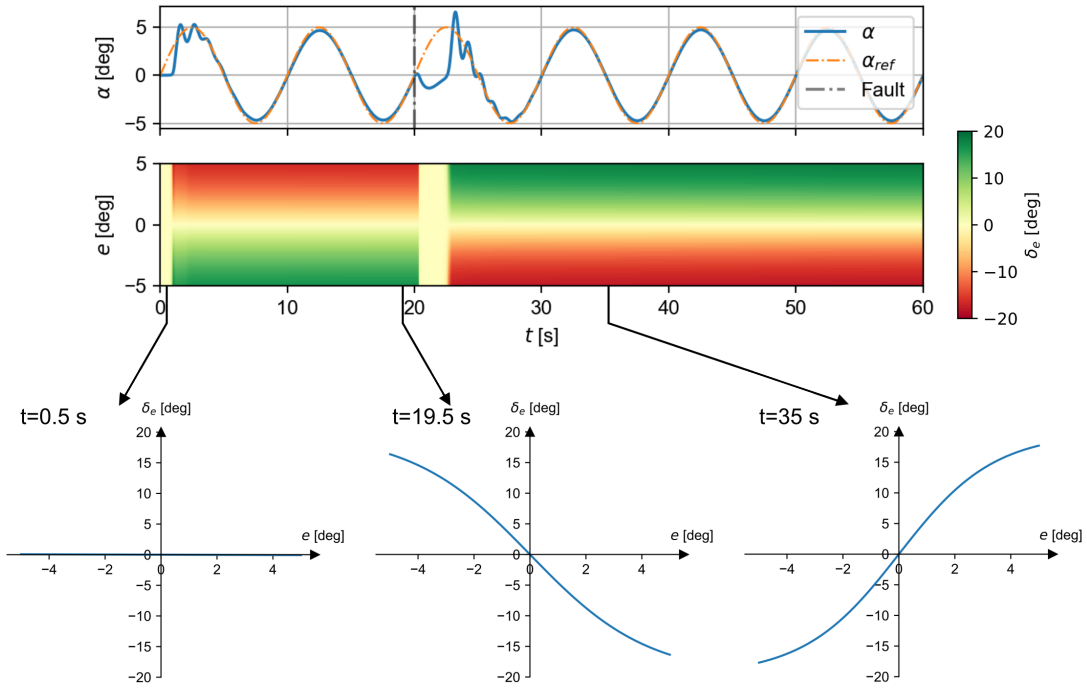
variation clearly, a scalar field plotted as a colour map is used. Specifically, at every time step, the policy function over the domain $e = [-5, 5]$ deg is recorded, by determining what the elevator deflection is over this domain of errors. This record is done at every single timestep, and the result can be plotted as a scalar field where the x and y axes are t and e, and the z axis is δ_e. The z axis is then mapped to a range of colours to represent the magnitude of elevator deflection.

These actor colourmaps and the corresponding state time trace are shown in Figure 3.20 for an unstable and a stable controller in Figure 3.20a and Figure 3.20b respectively. To add further information, three snapshots of the policy function are presented in Figure 3.20a and Figure 3.20b as well, these snapshots show what the policy function looks like at the select time steps: a mapping of what elevator deflection the agent will command for a given tracking error.

Inspecting these two controllers, a clear difference between them. For a stable controller, the actor-network would map the tracking error to a smooth control action, as evident by the smooth colour transitions in and the smooth policy function shown in Figure 3.20b at $t = 19.5$ s. This smooth actor-network can also be seen in the unstable network as well, but only before the reversed elevator fault is triggered. After the elevator is reversed, the actor-network converges to a very abrupt policy for the unstable controller, where any input away from zero tracking error results in the agent commanding a full negative or positive elevator deflection. This policy function resembles a step function more than a $\tanh$ function. Whereas for the stable controller, the agent managed to arrive at a policy function which is as smooth as before the fault, with only the gradient of the function being flipped.



(a) IDHP(λ) unstable controller.



(b) MIDHP(λ) stable controller.

Figure 3.20: Comparison of stable and unstable controllers on the reversed elevator fault, in 3.20a and 3.20b, first the aircraft and reference α overtime is plotted, then the policy function plotted over time is plotted as a colour plot, finally three snapshots of the policy function are shown at $t = 0.5,\ 19.5,\ 35$ s.

The C_v plots for the group of e samples in the reversed elevator fault experiment are shown in Figure 3.15. Unlike the C_v plots of the previous two faults, the graphs are cut short by the number of runs with unstable controllers, i.e. the presented C_v plots only show the C_v of the runs with stable controllers. Observing this graph, the C_v values for all algorithms are not as stable or converged as the plots of

Figure 3.15 and Figure 3.18. The most unstable of the C_v plot is for the IDHP(λ) controllers, where only two stable controllers were found, and for which the group of e statistics are not very indicative for the ensemble transient errors for all IDHP(λ) controllers. Nonetheless, the present result can still be used to draw some conclusions about the eligibility trace augmentation for the reversed elevator fault, which is that this augmentation detracts from the tolerance of the controller to this fault.

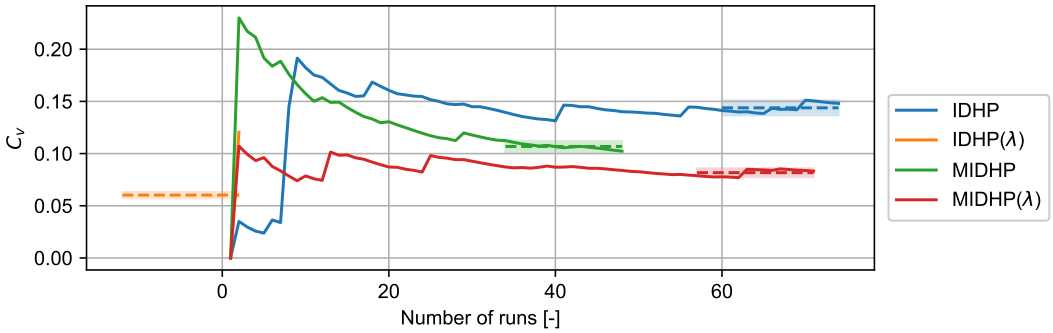


Figure 3.21: C_v plot for e under the reversed elevator fault in experiment 2, bounding boxes show mean of final 15 runs plus minus 5%.

The statistical testing results for the samples from this experiment are shown in Table 3.13.

Table 3.13: Statistical testing results of experiment 2 with the reversed elevator fault, p-values indicating statistically significant and insignificant differences shown in green and red respectively.

Statistical tests			
	IDHP(λ) vs IDHP	MIDHP vs IDHP	MIDHP(λ) vs IDHP
p-value	0.1213	0.3649	0.2036
A-value	1.0	1.0	1.0

3.7. Conclusion

This chapter demonstrated the effect of incorporating the IDHP algorithm with the three augmentations proposed: accumulating traces, multi-step update, and replacing traces plus multi-step update.

Prior to discussing the results, the method for incorporating the proposed augmentations with IDHP was presented. The MDP concerning flight control of the PH-LAB was also presented, along with explanations and specifications of what flight dynamics models were to be used, considerations for how to design the MDP variables and thus what kind of flight controller is to be designed. These sections help with answering **question 2** from the research questions.

The effects of the augmentations on the behaviour of IDHP were presented and discussed in the study of weight gradients. This showed that some differences are present in the actor and critic updates of the agent depending on what augmentations were used. These differences came in the form of larger magnitude updates, quicker updates, and more persistent updates compared to the updates found in the baseline IDHP algorithm.

In the first Monte Carlo experiment, the settling time and final error results from the four algorithms in a step tracking control task were presented. The two metrics used gave some level of indication of the simulation time needed for the agents to converge to their final policy, and what the quality of this final policy was. From the gathered results, it was observed that the proposed augmentations generally converged to a policy with more optimal tracking performance than the baseline algorithm. However, the time needed for IDHP augmented with eligibility traces to converge was slightly longer than IDHP. That said, the two other augmentations were found to have a smaller settling time, and thus indicating

they could converge to their final policy quicker.

In the second Monte Carlo experiment, the transient error results of the four algorithms in a sinusoidal tracking task with faults were presented. From the CG shift and damaged elevator fault scenarios, the augmentations generally showed slightly lower transient error than the baseline algorithm, except when IDHP is only augmented with multi-step TD updates which had a slightly higher transient error. In the reversed elevator fault, it was found that all augmentations led to a poorer rate of the algorithm arriving at a stable controller, with only the double augmentation of eligibility traces and multi-step TD updates giving somewhat similar rates of stable controllers. In all augmentations, the transient error of stable controllers was poorer than that of the baseline IDHP algorithm.

With these various studies, answers have been given for **question 3a, 3b, 3c, 3d** from the research questions. That being said, the answers thus provided are produced on flight control of the short-period model of the PH-LAB, and more detailed flight models can be used to provide observations on how the augmented algorithms will handle the real aircraft. Therefore, further research is needed.

Finally, the A-value for the gathered metrics from the conducted experiments is used to rank the four algorithms for each metric and experiment, and the resulting ranks are shown in Table 3.14. These ranks are decided based on which algorithm had the lowest A-value for the concerned metric, where the lowest A-value would have a rank of 1, and the highest A-value would have a rank of 4. The A-value of IDHP for all metrics is set to be 0.5, which stands for no difference in the sample values with IDHP. Note that the ranking for the reversed elevator experiment is done using how many runs out of the 100 conducted Monte Carlo runs yielded a stable controller.

Table 3.14: Ranking of the four algorithms according to each metric, rank 1 is best, rank 4 is worst

	Experiment 1		Experiment 2		
	t_s	e_f	shifted CG e	damaged elevator e	reversed elevator
Rank 1	MIDHP	IDHP(λ)	MIDHP(λ)	IDHP(λ)	IDHP
Rank 2	MIDHP(λ)	MIDHP	IDHP(λ)	MIDHP(λ)	MIDHP(λ)
Rank 3	IDHP	MIDHP(λ)	IDHP	IDHP	MIDHP
Rank 4	IDHP(λ)	IDHP	MIDHP	MIDHP	IDHP(λ)

From the Table 3.14, it can be seen that the baseline algorithm overall had ranks 3 or 4 for all the metrics, with IDHP only coming 1st in the reversed elevator fault, thus demonstrating that the proposed augmentations yielded more fault tolerant and performant controllers. Regarding which augmented algorithm is the best, since MIDHP(λ) had ranked 1st in one metric, 2nd in three metrics, and 3rd in one metric, this algorithm is overall the best. However, IDHP(λ) can also arguably be said to be as good if not better, since it ranked 1st in two metrics, 2nd in one metric, but 4th in two metrics.

These rankings can be discussed with a little more nuance, MIDHP(λ) ranked as the best algorithm in the fault tolerance experiments, ranking 1st in the shifted CG fault case, and 2nd in both the damaged and reversed elevator fault cases. But in nominal flight, which looks at the settling time and final error of the algorithms, it can be seen that MIDHP performed best, as it ranked 1st in terms of settling time, and 2nd in terms of final error.

Thus, MIDHP(λ) yields the most fault tolerant controller, and MIDHP yields the most performant controller under nominal flight conditions.

Bibliography

[1] I. Savage, "Comparing the fatality risks in united states transportation across modes and over time," *Research in Transportation Economics*, vol. 43, no. 1, pp. 9–22, 2013, The Economics of Transportation Safety, ISSN: 0739-8859. DOI: https://doi.org/10.1016/j.retrec.2012.12.011.

[2] I. A. T. Association, *Loss of control in-flight accident analysis report*, Distributed by IATA, 2019.

[3] K. Dally and E.-J. Van Kampen, "Soft actor-critic deep reinforcement learning for fault tolerant flight control," in *AIAA SCITECH 2022 Forum*, 2022, p. 2078.

[4] S. Heyer, D. Kroezen, and E.-J. Van Kampen, "Online adaptive incremental reinforcement learning flight control for a cs-25 class aircraft," Jan. 2020. DOI: 10.2514/6.2020-1844.

[5] G. J. Balas, "Flight control law design: An industry perspective," *European Journal of Control*, vol. 9, no. 2-3, pp. 207–226, 2003.

[6] D. Lee, D. Fahey, A. Skowron, M. Allen, U. Burkhardt, Q. Chen, S. Doherty, S. Freeman, P. Forster, J. Fuglestvedt, A. Gettelman, R. De León, L. Lim, M. Lund, R. Millar, B. Owen, J. Penner, G. Pitari, M. Prather, R. Sausen, and L. Wilcox, "The contribution of global aviation to anthropogenic climate forcing for 2000 to 2018," *Atmospheric Environment*, vol. 244, p. 117 834, 2021, ISSN: 1352-2310. DOI: https://doi.org/10.1016/j.atmosenv.2020.117834.

[7] M. Klöwer, M. R. Allen, D. S. Lee, S. R. Proud, L. Gallagher, and A. Skowron, "Quantifying aviation's contribution to global warming," *Environmental Research Letters*, vol. 16, no. 10, p. 104 027, Nov. 2021. DOI: 10.1088/1748-9326/ac286e.

[8] J. Benad and R. Vos, "Design of a flying v subsonic transport," in *33rd Congress of the International Council of the Aeronautical Sciences*, 2022.

[9] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017. arXiv: 1712.01815.

[10] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," 2019. arXiv: 1808.00177.

[11] M. Szuster and Z. Hendzel, "Discrete globalised dual heuristic dynamic programming in control of the two-wheeled mobile robot," *Mathematical Problems in Engineering*, vol. 2014, p. 628 798, 2014. DOI: 10.1155/2014/628798.

[12] H. Modares, F. L. Lewis, and Z.-P. Jiang, "H_∞ Tracking control of completely unknown continuous-time systems via off-policy reinforcement learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 10, pp. 2550–2562, 2015. DOI: 10.1109/TNNLS.2015.2441749.

[13] S. Roshanravan and S. Shamaghdari, "Adaptive fault-tolerant tracking control for affine nonlinear systems with unknown dynamics via reinforcement learning," *IEEE Transactions on Automation Science and Engineering*, vol. 21, no. 1, pp. 569–580, 2024. DOI: 10.1109/TASE.2022.3223702.

[14] L. Mark and S. Richard. "The markov property." (2005), [Online]. Available: https://shorturl.at/brtJ6 (visited on 02/11/2024).

[15] E. L. Thorndike, *Animal intelligence*. New York, The Macmillan Company, 1911.

[16] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[17] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[18] A. Plaat, *Deep reinforcement learning, a textbook*. Springer Singapore, 2022.

[19] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, *Gymnasium*, Mar. 2023. DOI: 10.5281/zenodo.8127026.

[20] E. Torenbeek, *Synthesis of subsonic airplane design: an introduction to the preliminary design of subsonic general aviation and transport aircraft, with emphasis on layout, aerodynamic design, propulsion and performance*. Springer Science & Business Media, 2013.

[21] R. C. Nelson *et al.*, *Flight stability and automatic control*. WCB/McGraw Hill New York, 1998, vol. 2.

[22] M. Palermo and R. Vos, "Experimental aerodynamic analysis of a 4.6%-scale flying-v subsonic transport," in *AIAA Scitech 2020 Forum*, 2020, p. 2228.

[23] C. Liu, "Turboelectric distributed propulsion system modelling," 2013.

[24] L. Buşoniu, R. Babuvska, B. De Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*. CRC Press, 2010.

[25] W. T. Miller, R. S. Sutton, and P. J. Werbos, "A menu of designs for reinforcement learning over time," in *Neural Networks for Control*. 1995, pp. 67–95.

[26] Q. Wei, D. Liu, and H. Lin, "Value iteration adaptive dynamic programming for optimal control of discrete-time nonlinear systems," *IEEE Transactions on Cybernetics*, vol. 46, no. 3, pp. 840–853, 2016. DOI: 10.1109/TCYB.2015.2492242.

[27] F. L. Lewis and D. Vrabie, "Reinforcement learning and adaptive dynamic programming for feedback control," *IEEE Circuits and Systems Magazine*, vol. 9, no. 3, pp. 32–50, 2009. DOI: 10.1109/MCAS.2009.933854.

[28] F. L. Lewis, D. Vrabie, and K. G. Vamvoudakis, "Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers," *IEEE Control Systems Magazine*, vol. 32, no. 6, pp. 76–105, 2012. DOI: 10.1109/MCS.2012.2214134.

[29] X. Wang and X. Tian, "Value approximation with least squares support vector machine in reinforcement learning system," *Journal of Computational and Theoretical Nanoscience*, vol. 4, pp. 1290–1294, Nov. 2007. DOI: 10.1166/jctn.2007.013.

[30] D. Bertsekas, *Dynamic programming and optimal control: Volume I*. Athena scientific, 2012, vol. 4.

[31] Q. Wei, D. Liu, and X. Yang, "Infinite horizon self-learning optimal control of nonaffine discrete-time nonlinear systems," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 4, pp. 866–879, 2015. DOI: 10.1109/TNNLS.2015.2401334.

[32] M. Lin, B. Zhao, D. Liu, X. Liu, and F. Luo, "Generalized policy iteration-based reinforcement learning algorithm for optimal control of unknown discrete-time systems," in *33rd Chinese Control and Decision Conference (CCDC)*, 2021, pp. 3650–3655. DOI: 10.1109/CCDC52312.2021.9601467.

[33] Y. Zhou, E.-J. v. Kampen, and Q. Chu, "Nonlinear adaptive flight control using incremental approximate dynamic programming and output feedback," *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 2, pp. 493–496, 2017. DOI: 10.2514/1.G001762.

[34] Y. Zhou, E. v. Kampen, and Q. Chu, "Incremental approximate dynamic programming for nonlinear adaptive tracking control with partial observability," *Journal of Guidance, Control, and Dynamics*, vol. 41, pp. 2554–2567, 12 2018. DOI: 10.2514/1.g003472.

[35] B. Luo, D. Liu, T. Huang, X. Yang, and H. Ma, "Multi-step heuristic dynamic programming for optimal control of nonlinear discrete-time systems," *Information Sciences*, vol. 411, pp. 66–83, 2017, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2017.05.005.

[36] D. Wang, J. Wang, M. Zhao, P. Xin, and J. Qiao, "Adaptive multi-step evaluation design with stability guarantee for discrete-time optimal learning control," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 9, pp. 1797–1809, 2023. DOI: 10.1109/JAS.2023.123684.

[37] T. Li, D. Zhao, and J. Yi, "Heuristic dynamic programming strategy with eligibility traces," in *2008 American Control Conference*, 2008, pp. 4535–4540. DOI: 10.1109/ACC.2008.4587210.

[38] J. Ye, Y. Bian, B. Xu, Z. Qin, and M. Hu, "Online optimal control of discrete-time systems based on globalized dual heuristic programming with eligibility traces," in *2021 3rd International Conference on Industrial Artificial Intelligence (IAI)*, 2021, pp. 1–6. DOI: 10.1109/IAI53119.2021.9619346.

[39] D. Liu, S. Xue, B. Zhao, B. Luo, and Q. Wei, "Adaptive dynamic programming for control: A survey and recent advances," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 1, pp. 142–160, 2021. DOI: 10.1109/TSMC.2020.3042876.

[40] T. Hanselmann, L. Noakes, and A. Zaknich, "Continuous-time adaptive critics," *IEEE Transactions on Neural Networks*, vol. 18, no. 3, pp. 631–647, 2007. DOI: 10.1109/TNN.2006.889499.

[41] D. Prokhorov and D. Wunsch, "Adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, pp. 997–1007, 1997. DOI: 10.1109/72.623201.

[42] L. Yu, W. Liu, Y. Liu, and F. E. Alsaadi, "Learning-based t-shdp() for optimal control of a class of nonlinear discrete-time systems," *International Journal of Robust and Nonlinear Control*, vol. 32, no. 5, pp. 2624–2643, 2022. DOI: https://doi.org/10.1002/rnc.5847.

[43] D. Liu, X. Xiong, and Y. Zhang, "Action-dependent adaptive critic designs," in *IJCNN'01. International Joint Conference on Neural Networks Proceedings*, vol. 2, 2001, 990–995 vol.2. DOI: 10.1109/IJCNN.2001.939495.

[44] J. Ye, Y. Bian, B. Luo, M. Hu, B. Xu, and R. Ding, "Costate-supplement adp for model-free optimal control of discrete-time nonlinear systems," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 1, pp. 45–59, 2024. DOI: 10.1109/TNNLS.2022.3172126.

[45] D. Liu and Q. Wei, "Policy iteration adaptive dynamic programming algorithm for discrete-time nonlinear systems," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 3, pp. 621–634, 2014. DOI: 10.1109/TNNLS.2013.2281663.

[46] Y. Zhou, E.-J. Van Kampen, and Q. Chu, "Incremental model based heuristic dynamic programming for nonlinear adaptive flight control," Oct. 2016.

[47] Y. Zhou, E.-J. Van Kampen, and Q. Chu, "Launch vehicle adaptive flight control with incremental model based heuristic dynamic programming," Sep. 2017.

[48] G. Venayagamoorthy, R. Harley, and D. Wunsch, "Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator," *IEEE transactions on neural networks*, vol. 13, pp. 764–73, Feb. 2002. DOI: 10.1109/TNN.2002.1000146.

[49] D. Liu, Y. Xu, Q. Wei, and X. Liu, "Residential energy scheduling for variable weather solar energy based on adaptive dynamic programming," *IEEE/CAA Journal of Automatica Sinica*, vol. 5, no. 1, pp. 36–46, 2018. DOI: 10.1109/JAS.2017.7510739.

[50] J. Qiao, M. Zhao, D. Wang, and M. Li, "Action-dependent heuristic dynamic programming with experience replay for wastewater treatment processes," *IEEE Transactions on Industrial Informatics*, vol. PP, pp. 1–9, Jan. 2024. DOI: 10.1109/TII.2023.3344130.

[51] C. Mu, Z. Ni, C. Sun, and H. He, "Data-driven tracking control with adaptive dynamic programming for a class of continuous-time nonlinear systems," *IEEE Transactions on Cybernetics*, vol. 47, no. 6, pp. 1460–1470, 2017. DOI: 10.1109/TCYB.2016.2548941.

[52] Y. Zhou, E.-J. Van Kampen, and Q. Chu, "Incremental model based online dual heuristic programming for nonlinear adaptive control," *Control Engineering Practice*, vol. 73, pp. 13–25, Apr. 2018. DOI: 10.1016/j.conengprac.2017.12.011.

[53] B. Sun and E.-J. van Kampen, "Incremental model-based global dual heuristic programming with explicit analytical calculations applied to flight control," *Engineering Applications of Artificial Intelligence*, vol. 89, p. 103 425, 2020, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2019.103425.

[54] M. Fairbank, E. Alonso, and D. Prokhorov, "Simple and fast calculation of the second-order gradients for globalized dual heuristic dynamic programming in neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 10, pp. 1671–1676, 2012. DOI: 10.1109/TNNLS.2012.2205268.

[55] G. Yen and P. DeLima, “Improving the performance of globalized dual heuristic programming for fault tolerant control through an online learning supervisor,” *IEEE Transactions on Automation Science and Engineering*, vol. 2, no. 2, pp. 121–131, 2005. DOI: 10.1109/TASE.2005.844122.

[56] D. Liu, D. Wang, D. Zhao, Q. Wei, and N. Jin, “Neural-network-based optimal control for a class of unknown discrete-time nonlinear systems using globalized dual heuristic programming,” *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 3, pp. 628–634, 2012. DOI: 10.1109/TASE.2012.2198057.

[57] J. Yi, S. Chen, X. Zhong, W. Zhou, and H. He, “Event-triggered globalized dual heuristic programming and its application to networked control systems,” *IEEE Transactions on Industrial Informatics*, vol. 15, no. 3, pp. 1383–1392, 2019. DOI: 10.1109/TII.2018.2850001.

[58] Y. Zhou, “Efficient online globalized dual heuristic programming with an associated dual network,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 12, pp. 10 079–10 090, 2023. DOI: 10.1109/TNNLS.2022.3164727.

[59] L. Baird, “Residual algorithms: Reinforcement learning with function approximation,” in *Machine Learning Proceedings*, San Francisco: Morgan Kaufmann, 1995, pp. 30–37, ISBN: 978-1-55860-377-6. DOI: https://doi.org/10.1016/B978-1-55860-377-6.50013-X.

[60] A. G. Ivakhnenko, “Polynomial theory of complex systems,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-1, no. 4, pp. 364–378, 1971. DOI: 10.1109/TSMC.1971.4308320.

[61] S. Liang and R. Srikant, “Why deep neural networks for function approximation?” In *International Conference on Learning Representations*, 2016. DOI: 10.48550/arXiv.1610.04161.

[62] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989. DOI: 10.1162/neco.1989.1.4.541.

[63] A. Graves, “Generating sequences with recurrent neural networks,” *arXiv preprint*, 2014. DOI: 10.48550/arXiv.1308.0850.

[64] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, “Exploring the limits of language modeling,” *arXiv preprint*, 2016. DOI: 10.48550/arXiv.1602.02410.

[65] C. Farabet, C. Couprie, L. Najman, and Y. Lecun, “Learning hierarchical features for scene labeling,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915–1929, Aug. 2013. DOI: 10.1109/TPAMI.2012.231.

[66] A. Krizhevsky, I. Sutskever, and G. Hinton, “Imagenet classification with deep convolutional neural networks,” *Neural Information Processing Systems*, vol. 25, Jan. 2012. DOI: 10.1145/3065386.

[67] G. E. Dahl, D. Yu, L. Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 30–42, DOI: 10.1109/TASL.2011.2134090.

[68] G. E. Dahl, D. Yu, L. Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 30–42, 2012. DOI: 10.1109/TASL.2011.2134090.

[69] S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint*, 2016. DOI: 10.48550/arXiv.1609.04747.

[70] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” in *International conference on machine learning*, Pmlr, 2014, pp. 387–395.

[71] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint*, 2013. DOI: 10.48550/arXiv.1312.5602.

[72] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” *arXiv preprint*, 2017. DOI: 10.48550/arXiv.1710.02298.

[73] H. van Hasselt, A. Guez, and D. Silver, “Pedestrian detection with unsupervised multi-stage feature learning,” in *Proceedings of the AAAI conference on Artificial Intelligence*, vol. 30, 2016. DOI: 10.1609/aaai.v30i1.10295.

[74] H. Hasselt, “Double q-learning,” in *Advances in Neural Information Processing Systems*, vol. 23, Curran Associates, Inc., 2010.

[75] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” *arXiv preprint*, 2015. DOI: 10.48550/arXiv.1511.05952.

[76] Z. Wang, N. de Freitas, M. Lanctot, T. Schaul, M. Hessel, and H. van Hasselt, “Dueling network architectures for deep reinforcement learning,” *arXiv preprint*, 2016. DOI: 10.48550/arXiv.1511.06581.

[77] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” *arXiv preprint*, 2016. DOI: 10.48550/arXiv.1602.01783.

[78] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug. 1988, ISSN: 1573-0565. DOI: 10.1007/BF00115009.

[79] M. G. Bellemare, W. Dabney, and R. Munos, “A distributional perspective on reinforcement learning,” *arXiv preprint*, 2017. DOI: 10.48550/arXiv.1707.06887.

[80] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, “Distributional reinforcement learning with quantile regression,” *arXiv preprint*, 2017. DOI: 10.48550/arXiv.1710.10044.

[81] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, “Noisy networks for exploration,” *arXiv preprint*, 2017. DOI: 10.48550/arXiv.1706.10295.

[82] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine learning*, vol. 8, pp. 229–256, 1992.

[83] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization,” *arXiv preprint*, 2015. DOI: 10.48550/arXiv.1502.05477.

[84] S. Kullback and R. A. Leibler, “On information and sufficiency,” *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[85] M. P. Deisenroth, G. Neumann, J. Peters, *et al.*, “A survey on policy search for robotics,” *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[86] H. Bai, R. Cheng, and Y. Jin, “Evolutionary reinforcement learning: A survey,” *Intelligent Computing*, vol. 2, p. 0025, 2023.

[87] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[88] 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.

[89] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *International conference on machine learning*, PMLR, 2018, pp. 1861–1870.

[90] B. D. Ziebart, *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. Carnegie Mellon University, 2010.

[91] S. Skogestad and I. Postlethwaite, *Multivariable Feedback Control: Analysis and Design*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2005, ISBN: 0470011688.

[92] M. Voutilainen, L. Viitasaari, and P. Ilmonen, “Note on ar(1)-characterisation of stationary processes and model fitting,” *Modern Stochastics: Theory and Applications*, vol. 6, no. 2, pp. 195–207, 2019, ISSN: 2351-6046. DOI: 10.15559/19-VMSTA132.

[93] R. V. Jategaonkar and F. Thielecke, “Evaluation of parameter estimation methods for unstable aircraft,” *Journal of Aircraft*, vol. 31, no. 3, pp. 510–519, 1994. DOI: 10.2514/3.46523.

[94] J. C. Gibson, “Handling qualities for unstable combat aircraft,” in *ICAS, Congress, 15 th, London, England*, 1986, pp. 433–445.

[95] C. Kamali, A. Pashilkar, and J. Raol, “Real-time parameter estimation for reconfigurablecontrol of unstable aircraft,” *Defence Science Journal*, vol. 57, no. 4, p. 381, 2007.

[96] A. K. Kundu, *Aircraft design*. Cambridge University Press, 2010, vol. 27.

[97] E. Torenbeek, *Synthesis of subsonic airplane design: an introduction to the preliminary design of subsonic general aviation and transport aircraft, with emphasis on layout, aerodynamic design, propulsion and performance*. Springer Science & Business Media, 2013.

[98] P. Abbeel, A. Coates, M. Quigley, and A. Ng, "An application of reinforcement learning to aerobatic helicopter flight," *Advances in neural information processing systems*, vol. 19, 2006.

[99] F. Fei, Z. Tu, J. Zhang, and X. Deng, "Learning extreme hummingbird maneuvers on flapping wing robots," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 109–115.

[100] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," *arXiv preprint arXiv:1707.06347*, 2016.

[101] A. De Marco, P. M. D'Onza, and S. Manfredi, "A deep reinforcement learning control approach for high-performance aircraft," *Nonlinear Dynamics*, vol. 111, no. 18, pp. 17 037–17 077, 2023.

[102] C. Tang and Y.-C. Lai, "Deep reinforcement learning automatic landing control of fixed-wing aircraft using deep deterministic policy gradient," in *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2020, pp. 1–9. DOI: 10.1109/ICUAS48674.2020.9213987.

[103] A. Adetifa, P. Okonkwo, B. B. Muhammed, and D. Udekwe, "Deep reinforcement learning for aircraft longitudinal control augmentation system," *Nigerian Journal of Technology*, vol. 42, no. 1, pp. 144–151, 2023.

[104] P. Seres, C. Liu, and E.-J. van Kampen, "Risk-sensitive distributional reinforcement learning for flight control," *IFAC-PapersOnLine*, vol. 56, no. 2, pp. 2013–2018, 2023.

[105] M. Zahmatkesh, S. A. Emami, A. Banazadeh, and P. Castaldi, "Robust attitude control of an agile aircraft using improved q-learning," in *Actuators*, MDPI, vol. 11, 2022, p. 374.

[106] R. Enns and J. Si, "Helicopter trimming and tracking control using direct neural dynamic programming," *IEEE Transactions on Neural networks*, vol. 14, no. 4, pp. 929–939, 2003.

[107] S. Ferrari and R. F. Stengel, "Online adaptive critic flight control," *Journal of Guidance, Control, and Dynamics*, vol. 27, no. 5, pp. 777–786, 2004.

[108] H. Li, L. Sun, W. Tan, X. Liu, and W. Dang, "Incremental dual heuristic dynamic programming based hybrid approach for multi-channel control of unstable tailless aircraft," *IEEE Access*, vol. 10, pp. 31 677–31 691, 2022.

[109] J. H. Lee and E.-J. Van Kampen, "Online reinforcement learning for fixed-wing aircraft longitudinal control," in *AIAA Scitech 2021 Forum*, 2021, p. 0392.

[110] C. Teirlinck and E.-J. Van Kampen, "Hybrid soft actor-critic and incremental dual heuristic programming reinforcement learning for fault-tolerant flight control," in *AIAA SCITECH 2024 Forum*, 2024, p. 2406.

[111] R. Konatala, E.-J. Van Kampen, and G. Looye, "Reinforcement learning based online adaptive flight control for the cessna citation ii (ph-lab) aircraft," in *AIAA Scitech 2021 Forum*, 2021, p. 0883.

[112] J. Rao, J. Wang, J. Xu, and S. Zhao, "Optimal control of nonlinear system based on deterministic policy gradient with eligibility traces," *Nonlinear Dynamics*, vol. 111, no. 21, pp. 20 041–20 053, 2023. DOI: 10.1007/s11071-023-08909-6.

[113] S. Baldi, Z. Zhang, and D. Liu, "Eligibility traces and forgetting factor in recursive least-squares-based temporal difference," *International Journal of Adaptive Control and Signal Processing*, vol. 36, no. 2, pp. 334–353, 2022. DOI: https://doi.org/10.1002/acs.3282.

[114] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on Nuclear Science*, vol. 44, no. 3, pp. 1464–1468, 1997. DOI: 10.1109/23.589532.

[115] K. Doel, U. Ascher, and E. Haber, "The lost honor of ℓ 2 -based regularization," in Aug. 2013, pp. 181–203, ISBN: 978-3-11-028222-1. DOI: 10.1515/9783110282269.181.

[116] J. Mulder, W. van Staveren, J. van der Vaart, E. de Weerdt, C. de Visser, A. in 't Veld, and E. Mooij, *Flight Dynamics Lecture Notes*. Delft, Netherlands: Delft University of Technology, 2013.

[117] D. Slock and T. Kailath, "Numerically stable fast transversal filters for recursive least squares adaptive filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 1, pp. 92–114, 1991. DOI: 10.1109/78.80769.

[118] Y. Zhou, "Online reinforcement learning control for aerospace systems," Available at https://doi.org/10.4233/uuid:5b875915-2518-4ec8-a1a0-07ad057edab4, PhD thesis, Delft University of Technology, Delft, The Netherlands, Apr. 2018.

[119] S. S. Haykin, *Adaptive filter theory*. Pearson Education India, 2002.

[120] D. Gupta, "Applicability of momentum in the methods of temporal learning," 2020.

[121] B. D. Nichols, "A comparison of eligibility trace and momentum on sarsa in continuous state-and action-space," in *2017 9th Computer Science and Electronic Engineering (CEEC)*, IEEE, 2017, pp. 55–59.

[122] D. Lee and N. He, "Target-based temporal-difference learning," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Jun. 2019, pp. 3713–3722.

[123] R. Rojas and R. Rojas, "The backpropagation algorithm," *Neural networks: a systematic introduction*, pp. 149–182, 1996.

[124] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[125] P. Cichosz, "Truncating temporal differences: On the efficient implementation of td (lambda) for reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 2, pp. 287–318, 1994.

[126] S. P. Singh and R. S. Sutton, "Reinforcement learning with replacing eligibility traces," *Machine learning*, vol. 22, no. 1, pp. 123–158, 1996.

[127] H. van Seijen, "Effective multi-step temporal-difference learning for non-linear function approximation," *arXiv preprint arXiv:1608.05151*, 2016.

[128] T. Kobayashi, "Adaptive and multiple time-scale eligibility traces for online deep reinforcement learning," *Robotics and Autonomous Systems*, vol. 151, p. 104 019, 2022, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2021.104019.

[129] F. Ballio and A. Guadagnini, "Convergence assessment of numerical monte carlo simulations in groundwater hydrology," *Water resources research*, vol. 40, no. 4, 2004.

[130] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[131] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.