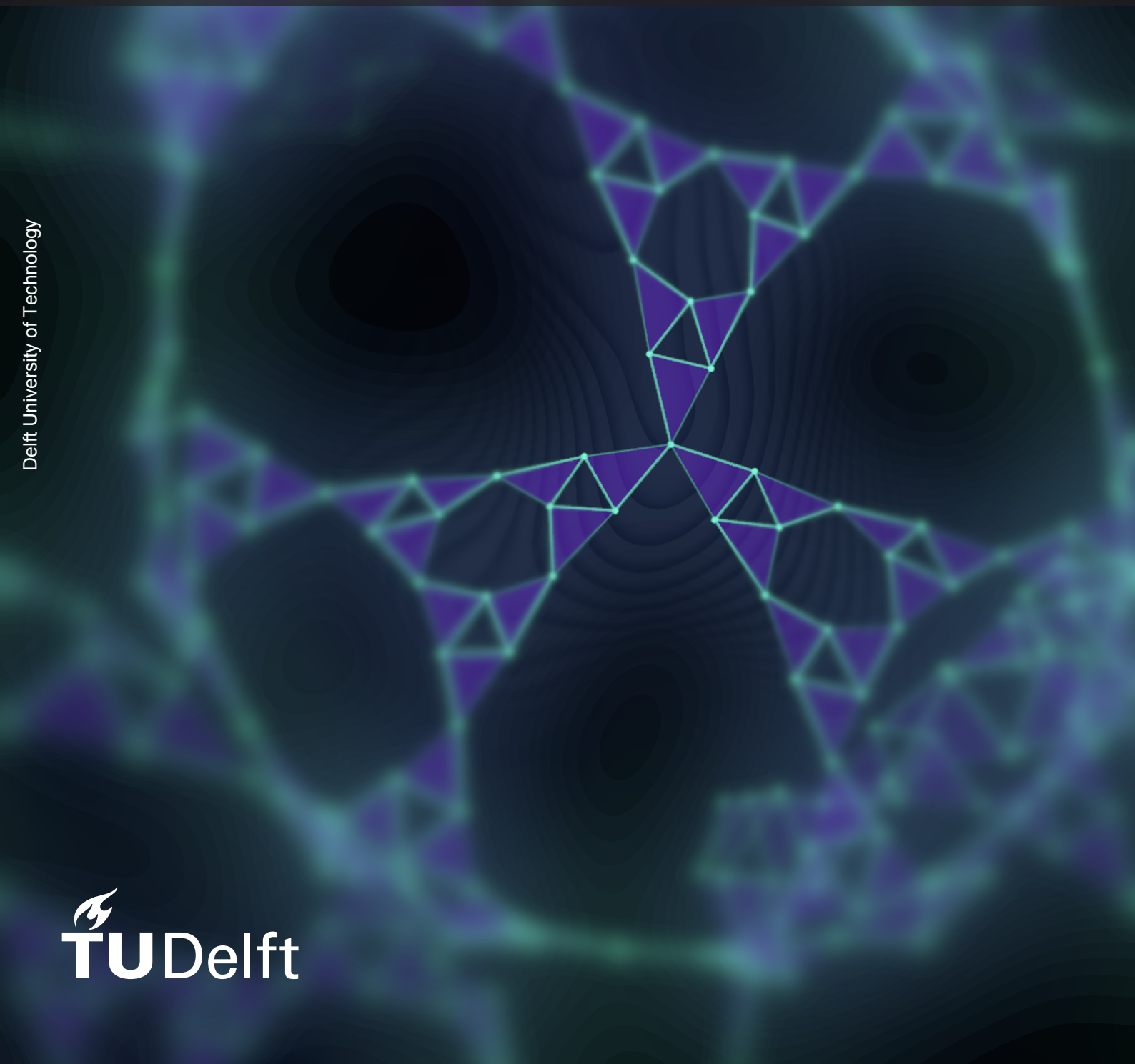
Reinforcement Learning for Flight Control

Intelligent Fault Tolerant Control of the Flying-V

Wing Chan



Delft University of Technology

TU Delft

Reinforcement Learning for Flight Control

Intelligent Fault Tolerant Control of the Flying-V

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	$[-, \text{s}]$
$p()$	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	$[-]$
$\boldsymbol{\theta}$	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
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
VLM	Vortex Lattice Method
CFD	Computational Fluid Dynamics

1 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-, the research efforts being the Flying-V, and finally the field of integrating reinforcement learning with flight control.

This chapter is laid out as follows. Section 1.1 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 1.2, and deep reinforcement learning discussed in Section 1.3. These two sections will start with touching on the earlier and more fundamental algorithms, moving progressively to the respective state of the art. Section 1.4 then introduces the technical background for the Flying-V, the aircraft which this research is concerned with. The problem of applying reinforcement learning to flight control is then discussed in Section 1.5, which will present various research efforts towards this goal and interesting directions for further study.

1.1. 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 [1], and a robotic arm can learn hand dexterity and mimic the hand movements of a human [2].

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.

1.1.1. 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 1.1.

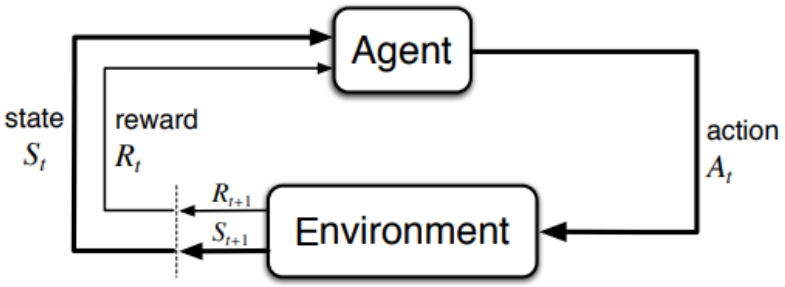


Figure 1.1: Flow diagram of the agent-environment interaction central to the MDP

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:

$$\mathcal{T} = S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T, S_T, A_T \tag{1.1}$$

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*:

$$(S_t, A_t, R_{t+1}, S_{t+1}) \tag{1.2}$$

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 1.3. 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.

$$p(s', r|s, a) \doteq Pr\{St = s', R_t = r|S_{t-1} = a, A_{t-1} = a\} \tag{1.3}$$

From Equation 1.3, it is possible to compute the expected reward for any state-action pairs $r(s, a)$:

$$r(s, a) \doteq \mathbb{E}\{R_t|S_{t-1} = s, A_{T-1} = a\} = \sum_r r \sum_{s'} p(s', r|s, a) \tag{1.4}$$

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 [3]–[5]. 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 1.5.

$$Pr\{St, R_t|S_{t-1}, A_{t-1}\} = Pr\{St, R_t|S_1, \ldots, S_{t-1}, A_1, \ldots, A_{t-1}\} \tag{1.5}$$

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 [6], which should be sufficient for a learner to decide what actions to take in order to enter into a trajectory which maximizes its' 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 1.1 can be considered equivalent to the control vector, the state vector, and an output vector in the state space formulation respectively.

1.1.2. 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 behavior adapting to receive more desirable stimuli in the context of domestic animal training, a so-called ”Law of Effect”[7]. 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 1.6 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”.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1.6}$$

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.

1.1.3. 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:

$$\pi(a|s) \doteq Pr\{A_t = a|S_t = s\} \tag{1.7}$$

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.

1.1.4. 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:

$$v_\pi(s) \doteq \mathbb{E}_\pi\{G_t|S_t = s\} \tag{1.8}$$

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:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi\{G_t|S_t = s, A_t = a\} \tag{1.9}$$

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.

1.1.5. Bellman Equation

The equation commonly referred to as *the* Bellman equation is the Bellman equation for the state-value function, presented in Equation 1.10, which is derived starting from the definition of the state-value function Equation 1.8:

$$\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi\{G_t|S_t = s\} \\
&= \mathbb{E}_\pi\{R_{t+1} + \gamma G_{t+1}|S_t = s\} \\
&= \sum_{r,g_{t+1}} p(r, g_{t+1}|s)[r + \gamma g_{t+1}] \\
&= \sum_{a,s'} \sum_{r,g_{t+1}} p(a, s', r, g_{t+1}|s)[r + \gamma g_{t+1}] \\
&= \sum_a \sum_{s',r} \sum_{g_{t+1}} \underbrace{p(a|s)}_{=\pi(a|s)} p(s', r|s, a) \underbrace{p(g_{t+1}|s', r, s, a)}_{=p(g_{t+1}|s'),\ \text{Markov}}[r + \gamma g_{t+1}] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{g_{t+1}} p(g_{t+1}|s')g_{t+1}] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]
\end{aligned} \tag{1.10}$$

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 1.10 is recursively defined. This recursive formulation lends itself readily to dynamic programming techniques, which simply means that Equation 1.10 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 1.10 was first derived by Richard Bellman and is associated with the basic foundations of the field of dynamic programming [8].

There also exists a Bellman equation for the action-value function, which is presented in Equation 1.11.

$$q_\pi(s, a) \doteq \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{1.11}$$

1.1.6. 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 distinguising 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 is generating the transition tuples is called the *behavioour policy*, while the policy using the generated tuples as training samples is called the *target policy* [9].

To seperate 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 behaviur 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 sipler algorithm, which converges to a stable policy much quicker albeit with slightly less optimal agent [10].

Model-Based vs Model-Free

The second characteristics 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 [11]. And 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 especially 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. And when models of the environment are not at hand, an algorithm can be designed in 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-bsed 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 uses the approximator in the value function and the policy function respectively.

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

1.1.7. 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 1.2.

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.

$$Q(s_t, s_t) = Q(s_t, a_t) + \alpha[G_t - Q(s_t, a_t)] \tag{1.12}$$

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:

$$Q(s_t, s_t) = Q(s_t, a_t) + \alpha[R + \gamma Q(s_{t+1}, a_{s+1}) - Q(s_t, a_t)] \tag{1.13}$$

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 [9]. 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.

1.2. 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[12]. 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[8]. In the context of reinforcement learning, classical applications of dynamic programming revolve around the Bellman equation stated in Equation 1.10 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 1.2.1. Modern applications of dynamic programming extend the method of policy iteration to make it more computationally tractable and is described in Section 1.2.3.

The classes of algorithms that are branched off from dynamic programming methods are summarized in Figure 1.2.

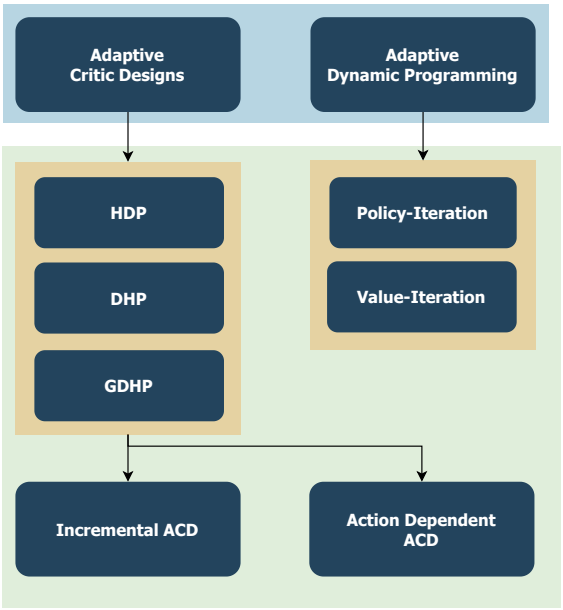


Figure 1.2: Overview of Approximate Dynamic Programming and Adaptive Critic Design algorithms.

1.2.1. 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 1.3 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.

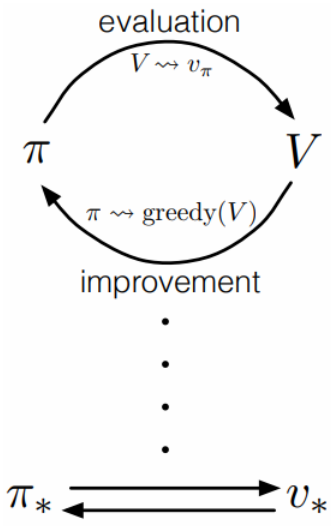


Figure 1.3: The method of policy iteration, by iteratively evaluation a policy's value function, and subsequently determining the greedy policy for that value function, the policy and value function eventually converges to a fixed iteration point when they are optimal for the given MDP.

This algorithm is proven to converge toward the optimal value function and policy[9] but with some caveats, namely, it assumes environment knowledge and the curse of dimensionality which will be described further in this text. 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 1.14. 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.

$$v_{\pi,k+1} = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi,k}(s')] \quad \forall s \in \mathcal{S} \tag{1.14}$$

Note that to compute the update formulated in Equation 1.14, 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 1.10, it can be seen that the update equation Equation 1.14 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:

$$\pi'(a|s) = \begin{cases} 1, & \text{if } a = \underset{a}{\text{argmax}}\ q_\pi(s, a) \\ 0, & \text{otherwise} \end{cases} \tag{1.15}$$

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 [8]. 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[9].

1.2.2. 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 1.14 and Equation 1.15. 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 [9], who they 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 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[13], [14].

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.

1.2.3. 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 that exist, which means having a specific value and a specific action probability distribution defined for each state becomes impractical. This is the so-called *curse of dimensionality*, alluded to in Section 1.2.1. Moreover, evaluation of the Bellman equations also requires a known transition probability distribution of the environment. Not only does this also 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 [15]

1: **Initialize**. Define a policy $\pi_0(\mathbf{x}_k)$ which is admissible, i.e. stabilizing, and an arbitrary initial value function $V_0(\mathbf{x})$.

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

$$V_{i+1}(\mathbf{x}_t) = r(\mathbf{x}_t, \pi_i(\mathbf{x}_t)) + \gamma V_{i+1}(\mathbf{x}_{t+1}) \tag{1.16}$$

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

$$\pi_{i+1}(\mathbf{x}_t) = \underset{\pi(\mathbf{x}_t)}{\operatorname{argmin}}[r(\mathbf{x}_t, \pi_i(\mathbf{x}_t)) + \gamma V_i(\mathbf{x}_{t+1})] \tag{1.17}$$

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

Algorithm 1 fits into the mold of generalized policy iteration, which incorporates certain aspects of optimal control theory and has some note-worthy aspects. Firstly, the policy evaluation step Equation 1.14 of this algorithm is not formulated in the MDP framework like in Equation 1.14, and instead is the Hamilton-Jacobi-Bellman (HJB) equation which is generally difficult to evaluate especially online[16] but can be solved approximately by methods such as least squares approximation using environment observations[17], or by iteratively evaluating the equation until convergence[15]. 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 1.18 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 1.19, then it is known that the policy improvement step takes the form of Equation 1.20[18].

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t) + g(\mathbf{x}_t)\mathbf{u}_t \tag{1.18}$$

$$r(\mathbf{x}_t, \mathbf{u}_t) = \mathbf{x}_t^\top \mathbf{Q}\mathbf{x}_t + \mathbf{u}_t^\top \mathbf{R}\mathbf{u}_t \tag{1.19}$$

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

$$\pi_{i+1}(\mathbf{x}_t) = -\frac{\gamma}{2}\mathbf{R}^{-1} g^\top(\mathbf{x}_t) \frac{\partial V_i(\mathbf{x}_{t+1})}{\partial \mathbf{x}_{t+1}} \tag{1.20}$$

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 [15]

1: **Initialize**. Define an arbitrary policy $\pi_0(\mathbf{x}_k)$, and an arbitrary initial value function $V_0(\mathbf{x})$.
2: **Policy Evaluation**. Determine the value function of the current policy using the Hamilton-Jacobi-Bellman Equation:

$$V_{i+1}(\mathbf{x}_t) = r(\mathbf{x}_t, \pi_i(\mathbf{x}_t)) + \gamma V_i(\mathbf{x}_{t+1}) \tag{1.21}$$

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

$$\pi_{i+1}(\mathbf{x}_t) = \underset{\pi(\mathbf{x}_t)}{\operatorname{argmin}}[r(\mathbf{x}_t, \pi_i(\mathbf{x}_t)) + \gamma V_i(\mathbf{x}_{t+1})] \tag{1.22}$$

4: If $V_i(\mathbf{x}) \equiv V_{i-1}(\mathbf{x})$ 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 1.21, 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. [19] 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. [20] 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. [21], [22] 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 HJB equation can 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 binary situation can be reconciled 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 [9].

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 [23].

A high-level view of the multi-step ADP algorithm is shown in Algorithm 3, with n being the number of steps. To make the choice of n automatic, Wang et al. [24] 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 [23].

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

$$V_{i+1}(\mathbf{x}_t) = \gamma^n V_i(\mathbf{x}_{t+1}) + \sum_{l=t}^{t+n-1} \gamma^{l-t} r(\mathbf{x}_l, \pi_i(\mathbf{x}_l)) \tag{1.23}$$

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

$$\pi_{i+1}(\mathbf{x}_t) = \underset{\pi(\mathbf{x}_t)}{\operatorname{argmin}}[r(\mathbf{x}_t, \pi_i(\mathbf{x}_t)) + \gamma V_i(\mathbf{x}_{t+1})] \tag{1.24}$$

4: If $V_i(\mathbf{x}) \equiv V_{i-1}(\mathbf{x})$ 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 [25], [26]. 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 1.25, where $\boldsymbol{\theta}(t)$ 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 $\Delta\boldsymbol{\theta}(t)$ is the update made to the function approximator at time t.

$$\begin{aligned} \boldsymbol{\theta}(T+1) &= \boldsymbol{\theta}(t) + \mathcal{E}(t) \\ \mathcal{E}(t+1) &= \mathcal{E}(t) + \nabla\boldsymbol{\theta}(t), \quad \mathcal{E}(0) = 0 \end{aligned} \tag{1.25}$$

1.2.4. 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 [27]–[29], notably with the progenitor of ACD Werbos also using these terms and reinforcement learning interchangeably [13]. 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 [29] 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 [23], [27], [30].

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 [31]. 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.

$$V(\mathbf{x}_t) = r_t + \gamma V(\mathbf{x}_{t+1}) \tag{1.26}$$

$$r_t = \frac{1}{2}(\mathbf{x}_t - \mathbf{x}_{t,ref})^\top (\mathbf{x}_t - \mathbf{x}_{t,ref}) \tag{1.27}$$

Heuristic Dynamic Programming

HDP is characterized by using the critic to estimate the value function $V(\mathbf{x}_t)$ itself:

This critic is evaluated by the critic TD error $\mathbf{e}_c(t)$ defined in Equation 1.28, and is trained to minimize the critic error function Equation 1.29. To perform training, the function approximator of the critic is updated using a gradient descent method to minimize the error function, meaning the parameters $\boldsymbol{\theta}(t)$ are updated according to Equation 1.30.

$$\mathbf{e}_{1,c}(t) = V(\mathbf{x}_t) - r_t - \gamma V(\mathbf{x}_{t+1}) \tag{1.28}$$

$$\mathbf{E}_c(t) = \frac{1}{2}\mathbf{e}_{1,c}(t)^\top \mathbf{e}_{1,c}(t) \tag{1.29}$$

$$\boldsymbol{\theta}_c(t+1) = \boldsymbol{\theta}_c(t) - \nabla\boldsymbol{\theta}_c(t) \tag{1.30}$$

$$\text{Where} \quad \nabla\boldsymbol{\theta}_c(t) = \eta_c \frac{\partial \mathbf{E}_c(T)}{\partial \boldsymbol{\theta}_c(t)} = \eta_c \frac{\partial \mathbf{E}_c(T)}{\partial \mathbf{e}_{1,c}(t)} \frac{\partial \mathbf{e}_{1,c}(t)}{\partial V(\mathbf{x}_t)} \frac{\partial V(\mathbf{x}_t)}{\partial \boldsymbol{\theta}_c(t)}$$

Correspondingly, there is also the actor loss, the actor error function, and the actor update rule that are expressed in the following equations:

$$\mathbf{e}_a(t) = V(\mathbf{x}_t) \tag{1.31}$$

$$\mathbf{E}_a(t) = \frac{1}{2}\mathbf{e}_a(t)^\top \mathbf{e}_a(t) \tag{1.32}$$

$$\boldsymbol{\theta}_a(t+1) = \boldsymbol{\theta}_a(t) - \nabla\boldsymbol{\theta}_a(t) \tag{1.33}$$

$$\text{Where} \quad \nabla\boldsymbol{\theta}_a(t) = \frac{\partial \mathbf{E}_a(t+1)}{\partial \boldsymbol{\theta}_a(t)} = \frac{\partial \mathbf{E}_a(t+1)}{\partial \mathbf{e}_a(t+1)} \frac{\partial \mathbf{e}_a(t+1)}{\partial V(\mathbf{x}_{t+1})} \frac{\partial V(\mathbf{x}_{t+1})}{\partial \mathbf{x}_{t+1}} \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_t} \frac{\partial \mathbf{u}_t}{\partial \boldsymbol{\theta}_a(t)} \tag{1.34}$$

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 [23], [32], [33].

Observing the update equations, it can be seen that the actor update contains the partial derivative $\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_t}$, this is a derivative that needs to be obtained using a system model which would define the relation between the state $\mathbf{x}_{t+1}$ and the action $\mathbf{u}_t$. 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 $V(\mathbf{x}_t, \mathbf{u}_t)$, this allows the chain rule expansion in Equation 1.34 to reduce the term $\frac{\partial V(\mathbf{x}_{t+1})}{\partial \mathbf{x}_{t+1}} \frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_t}$ to only $\frac{\partial V(\mathbf{x}_{t+1})}{\partial \mathbf{u}_t}$ since V would be an explicit function of $\mathbf{u}_t$, thus eliminating the system dynamics $\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_t}$.

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 [34], and a launch vehicle control task [35]. In this case, the partial $\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{u}_t}$ 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 [29], [36], but nonetheless have been deployed successfully [37]–[39].

Dual Heuristic Programming

DHP is characterized by the critic estimating the gradient of the value function $\lambda(\mathbf{x}_t)$, instead of the value function directly. This gradient can also be referred to as the *costate* [15]:

$$\lambda(\mathbf{x}_t) = \frac{\partial V(\mathbf{x}_t)}{\partial \mathbf{x}_t} \tag{1.35}$$

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 1.36, with the critic error function and function parameter update remaining unchanged.

$$\mathbf{e}_{2,c}(t) = \lambda(\mathbf{x}_t) - \frac{\partial r_t}{\partial \mathbf{x}_t} - \gamma\lambda(\mathbf{x}_{t+1})\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t} \tag{1.36}$$

$$\mathbf{E}_c(t) = \frac{1}{2}\mathbf{e}_{2,c}(t)^\top \mathbf{e}_{2,c}(t) \tag{1.37}$$

$$\boldsymbol{\theta}_c(t+1) = \boldsymbol{\theta}_c(t) - \nabla\boldsymbol{\theta}_c(t) \tag{1.38}$$

$$\text{Where} \quad \nabla\boldsymbol{\theta}_c(t) = \eta_c \frac{\partial \mathbf{E}_c(T)}{\partial \boldsymbol{\theta}_c(t)} = \eta_c \frac{\partial \mathbf{E}_c(T)}{\partial \mathbf{e}_{2,c}(t)} \frac{\partial \mathbf{e}_{2,c}(t)}{\partial \lambda(\mathbf{x}_t)} \frac{\partial \lambda(\mathbf{x}_t)}{\partial w_c(t)}$$

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 $\frac{\partial \mathbf{x}_{t+1}}{\partial \mathbf{x}_t}$ 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 Incremental Dual Heuristic Programming (IDHP). Application of IDHP to the task of flight control demonstrated improved reference tracking performance and fault tolerance than a pure DHP algorithm [40], [41], 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 $\begin{bmatrix} V(\mathbf{x}_t) & \lambda(\mathbf{x}_t) \end{bmatrix}^\top$.

This results in the GDHP critic error function being composed of two terms, a HDP and a DHP td error

$$\mathbf{E}_c(t) = \frac{1}{2}\mathbf{e}_{1,c}(t)^\top \mathbf{e}_{1,c}(t) + \frac{1}{2}\mathbf{e}_{2,c}(t)^\top \mathbf{e}_{2,c}(t) \tag{1.39}$$

$$\boldsymbol{\theta}_c(t+1) = \boldsymbol{\theta}_c(t) - \nabla\boldsymbol{\theta}_c(t) \tag{1.40}$$

$$\text{Where} \quad \nabla\boldsymbol{\theta}_c(t) = \eta_c \frac{\partial \mathbf{E}_c(T)}{\partial \boldsymbol{\theta}_c(t)} = \eta_c \left(\frac{\partial \mathbf{E}_c(T)}{\partial \mathbf{e}_{1,c}(t)} \frac{\partial \mathbf{e}_{1,c}(t)}{\partial V(\mathbf{x}_t)} \frac{\partial V(\mathbf{x}_t)}{\partial \boldsymbol{\theta}_c(t)} + \frac{\partial \mathbf{E}_c(T)}{\partial \mathbf{e}_{2,c}(t)} \frac{\partial \mathbf{e}_{2,c}(t)}{\partial \lambda(\mathbf{x}_t)} \frac{\partial \lambda(\mathbf{x}_t)}{\partial \boldsymbol{\theta}_c(t)} \right) \tag{1.41}$$

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 [42], [43] and the partial derivative $\frac{\partial \lambda(x_t)}{\partial w_c(t)}$ from Equation 1.41 is then computed by replacing it with the partial derivative $\frac{\partial^2 V(x_t)}{\partial x_t \partial w_c(t)}$, 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 [44]–[46], 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 [47] proposed the idea of building the critic as two function approximations but with a novelty of formulating the $\lambda(x_t)$ approximator based explicitly on the $V(x_t)$ 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.

1.3. 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 [48] 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 1.4. 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.

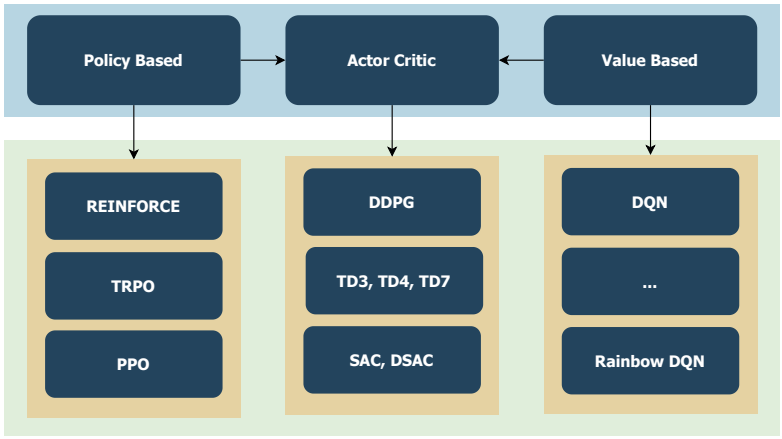


Figure 1.4: Overview of deep reinforcement learning algorithms

This section is laid out as follows, first a description of deep learning is given in Section 1.3.1, then value-based methods are discussed in Section 1.3.2, followed by policy-based methods in Section 1.3.3, and lastly a discussion on the combination of policy and value-based methods called actor-critic methods in Section 1.3.4.

1.3.1. 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 [49]. The large network structure of DNNs allows them to be trained more efficiently to approximate a given function [50], 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 [51], 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 [52] and language modeling [53].

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 [54], [55], speech recognition [56], [57], and in handwriting generation [52]. 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 $f(\mathbf{x})$ in the following manner:

$$f(\mathbf{x}) = g(\boldsymbol{\theta}\mathbf{x} + \mathbf{b}) \tag{1.42}$$

With $\boldsymbol{\theta}$ a square weight matrix, $\mathbf{b}$ 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:

$$\mathbf{y} = h(\mathbf{x}) = f_1(f_2(\dots(f_k(\mathbf{x})))) \tag{1.43}$$

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 [58]. 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:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla J(\boldsymbol{\theta}_t) \tag{1.44}$$

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 [58].

Finally, when it comes to using DNNs as function approximators, one should keep in mind that they can break many of the pre-existing convergence properties that reinforcement learning algorithms would otherwise possess with a linear function approximator, or no function approximation at all [9], [59]. This practically does not mean doom, as long as proper hyperparameter tuning and addition of convergence/stability aiding measures are used, as will be made evident in subsequent sections.

1.3.2. 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:

$$\pi'(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \underset{a}{\text{argmax}}\ q_\pi(s,a) \\ \epsilon, & \text{otherwise} \end{cases}, \quad 0 < \epsilon < 1 \tag{1.45}$$

Deep Q Network

The first successful DRL algorithm came from the value-based approach, and it was the Deep Q Network (DQN) algorithm [60]. 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** [60]. 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** [60]. The target here is the value that a value estimate is moved towards, as described in Section 1.1.7. 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 [61]. These various modules are compiled in the following list:

- **Double Learning** [62]. 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 [63].
- **Prioritized Experience Replay** [64]. 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** [65]. A novel neural network architecture, named *dueling architecture*, is adopted to approximate the value function estimates. The dueling 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 dueling architecture, the complexity of learning for the neural network is reduced, which contributes towards higher sample efficiency and better agent performance.
- **Multi-step Bootstrapping** [9], [66]. 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 [67] and corroborated in ADP research [23], 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 [67]. This variation was used on four standard algorithms by Mnih et al. [66], 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** [68], [69]. 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 allows 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 [68].
- **Noisy networks** [70]. 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.

1.3.3. 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 1.3.4. The policy $\pi(a|s, \boldsymbol{\theta_a})$ in such methods is approximated with a DNN parameterized with the parameters $\boldsymbol{\theta_a}$. 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 1.10 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:

$$\nabla J(\boldsymbol{\theta_a}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta_a}) \tag{1.46}$$

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 [9].

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 [71]. 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 [9]:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta G_t \nabla \ln(\pi(A_t|S_t, \boldsymbol{\theta}_t)) \tag{1.47}$$

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 1.47 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 [9], [71]. 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 [72], 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 [73].

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

$$\begin{aligned} &\underset{\boldsymbol{\theta}_a}{\text{maximize}} \ \mathbb{E}\left[\frac{\pi(a|s,\boldsymbol{\theta}_a)}{\pi(a|s,\boldsymbol{\theta}_{a,old})} Q_{\pi_{old}}(s,a)\right] \\ &\text{subject to} \ \mathbb{E}\big[D_{KL}\big(\pi(\cdot|s,\boldsymbol{\theta}_a)||\pi(\cdot|s,\boldsymbol{\theta}_{a,old})\big)\big] \leq \delta \end{aligned} \tag{1.48}$$

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 [74] and evolutionary reinforcement learning [75]. The optimization problem from Equation 1.48 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 [76]. Hence, more augmentations are needed.

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 1.48, 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 1.49 with a tunable parameter β.

$$\underset{\boldsymbol{\theta}_a}{\text{maximize}} \ \mathbb{E}\left[\frac{\pi(a|s,\boldsymbol{\theta}_a)}{\pi(a|s,\boldsymbol{\theta}_{a,old})} A_{\pi_{old}}(s,a) - \beta D_{KL}[\pi(\cdot|s,\boldsymbol{\theta}_a)||\pi(\cdot|s,\boldsymbol{\theta}_{a,old})]\right] \tag{1.49}$$

In fact, Equation 1.49 is the optimization problem suggested by the theory which justifies TRPO [76]. 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 1.49. Such a surrogate objective function $J_{CLIP}(\boldsymbol{\theta}_a)$ is defined in Equation 1.50, 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.

$$\begin{aligned} \underset{\boldsymbol{\theta}_a}{\text{maximize}} \ J_{CLIP}(\boldsymbol{\theta}_a) =& \mathbb{E}\left[\min\left(r(\boldsymbol{\theta}_a), \text{clip}\big(r(\boldsymbol{\theta}_a); 1-\epsilon, 1+\epsilon\big)\right) A_{\pi_{old}}(s,a)\right] \\ \text{Where} \quad r(\boldsymbol{\theta}_a) =& \frac{\pi(a|s,\boldsymbol{\theta}_a)}{\pi(a|s,\boldsymbol{\theta}_{a,old})}, \quad A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \end{aligned} \tag{1.50}$$

Just like TRPO, PPO uses samples from generated trajectories to estimate the advantage function values. By using $J_{CLIP}(\boldsymbol{\theta}_a)$, 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 1.5.

1.3.4. 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 a big disadvantage of still being ham-strung by the curse of dimensionality when either

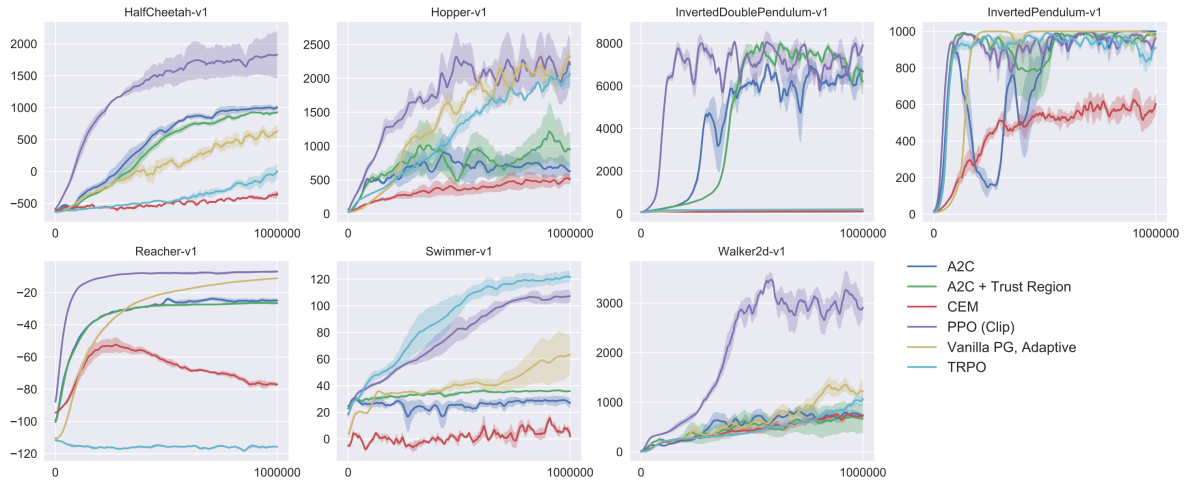


Figure 1.5: Comparison of PPO with TRPO and various other algorithms on several benchmark environments, taken from [76]

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 1.2.2, 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 [77].

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 [59]. 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 [59]. 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 1.51 which the critic is trained to minimize, and the policy

gradient Equation 1.53 used to update the actor.

$$J_c = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i, \boldsymbol{\theta}_c))^2 \tag{1.51}$$

$$y_i = r_i + \gamma Q(s_i, \pi(s_{i+1}, \boldsymbol{\theta}_a^T), \boldsymbol{\theta}_c^T) \tag{1.52}$$

$$\nabla_{\boldsymbol{\theta}_a} J_a = \frac{1}{N}\sum_i \nabla_a Q\big(s_i, \pi(s_i, \boldsymbol{\theta}_a), \boldsymbol{\theta}_c\big) \nabla_{\boldsymbol{\theta}_a} \pi(s_i, \boldsymbol{\theta}_a) \tag{1.53}$$

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

A noteworthy achievement of the DDPG algorithm is its ability to triumph in the face of 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 [77].

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 [63], 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 1.52 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:

$$y_i = r_i + \gamma Q\big(s_i, \pi(s_{i+1}, \boldsymbol{\theta}_a^T) + \epsilon, \boldsymbol{\theta}_c^T\big) \tag{1.54}$$

Soft Actor Critic

Around the same time but independent of the efforts of Fujimoto et al., Haarnoja et al. [78] 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* [79]. 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:

$$J(\pi) = \sum_t \mathbb{E}[r_t + \underbrace{\alpha \mathcal{H}(\pi(s_t))}_{\substack{\text{entropy of} \\ \text{policy}}}] \tag{1.55}$$

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 [63], 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.

1.4. Flying-V

The Flying-V is a novel design for jet-powered passenger airliners, it features a distinct V-shaped flying wing body which contains the passenger and cargo compartments while promising the potential for more sustainable aviation through efficient flight [80], [81]. This aircraft design will serve as the environment in the MDP that the developed reinforcement learning agent will control, and thus this section will serve to introduce the background on this aircraft and relevant notes of the design.

1.4.1. Flying-V Overview

The design of this aircraft focused on improving the efficiency of commercial jet airliners, which have a tube and wing configuration and present additional undesired parasitic drag from the tube fuselage. At cruising altitude, passenger cabins are pressurized to provide comfortable breathing conditions, consequently, passenger cabins are shaped like a cylinder as it is a structurally efficient geometry for pressurized vessels. By placing the passenger cabins along the span of a heavily swept-back wing, it is possible to use a cylindrical shape to retain this structural efficiency, in addition to more efficiently fitting the cabin inside an airfoil profile and thus reducing the cabin's parasitic drag. This serves as the primary design factor driving the design of the Flying V, resulting in the Λ shape of this aircraft's design [80], depicted in Figure 1.6.

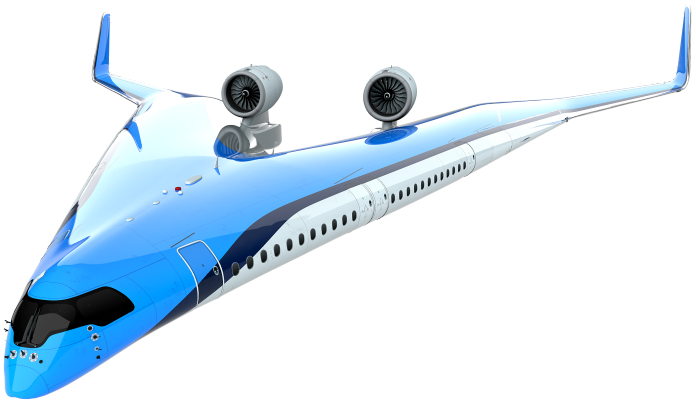


Figure 1.6: Artist's impression of the Flying V[1].

This configuration results in both aerodynamic and structural efficiency improvements when compared to the A350-900, which is in the same category of passenger capacity and flight range. Aerodynamically, the Flying-V has demonstrated a 10-% higher lift-to-drag ratio than the A350 [80], which allows it to carry more payload for the same extent of drag and thus ferry passengers more efficiently. Moreover, it has an estimated structural weight saving of 17% compared to an A350-like design [82], [83], not only making it more material-efficient but also requiring less lift to fly. Unlike contemporary airliners, the

[1] Retrieved from `https://www.tudelft.nl/en/ae/flying-v`

engine of the Flying-V is placed above the wings, which will shield the noise generated by the turbofan engines from the ground, and is expected to reduce noise pollution, especially during the terminal flight phases of takeoff and landing.

The Flying-V is a design which is being actively improved upon. Using a 4.6%-scale model of the aircraft for wind tunnel tests, Palermo and Vos experimentally deduced an optimal range of location to place the design's center of gravity to ensure longitudinal stability [84], which served as an important variable to base future design work upon. Engine positioning as well as its effect on the aerodynamics of the Flying-V were investigated through numerical simulations by Pascual and Vos [85], who noted the extreme detrimental impact of misplacing the engines which at worst resulted in a lift-to-drag ratio loss of 55%, and selected a position which compromised between minimal aerodynamic efficiency loss and controllability under a one-engine inoperative scenario. The aerodynamic effects of this engine placement was further investigated experimentally in a wind tunnel using the 4.6%-scale model by van Empelen and Vos [83], who showed that the engine installation resulted in a small increase of the lift-curve slope, and found an upper bound for the nose-down moment cause by engine thrust. The concept was then extended to a family of designs by Oosterom and Vos [86], who presented a three-member **FV-XXX** family concept, key characteristics of each member in the family are tabulated in Table 1.1.

Table 1.1: Flying-V family of designs[87].

Parameter	Unit	FV-800	FV-900	FV-1000
Wing span	$[m]$	55.5	60.7	65
Passengers	[-]	293	328	361
Design range	$[km]$	11,200	14800	15300
OEW	$[kg \times 10^3]$	99.9	115	129
MTOW	$[kg \times 10^3]$	185	234	266

The outboard wing geometry was optimized under a study by van Luijk and Vos [88], this optimization was performed with constraints on the area, wing planform limits, and the integration of elevon surfaces. This geometry optimization resulted in a wing design that increased the lift-to-drag ratio of the Flying-V by a further 8.4%.

1.4.2. Flight Control Challenges

The traditional tube-and-wing design of an aircraft has been widely adopted due to its inherently stable and easy to trim nature. The location of horzontal and vertical stabilizers all the way at the tail end of an aircraft allows these surfaces to provide a high degree of passive stabilizing moments, resulting in the yawing and pitching moment coefficients of such designs to reach stabilizing values easily. Their effects can be compared to that of the fins at the tail end of an arrow, which places the center of pressure aft of the center of gravity, which allows the aerodynamic forces to naturally steer the arrow into a straight flight.

In a flying wing design such as the Flying-V, this natural stability is largely forgone. Instead, the sweep of the wings needs to be sufficiently high to recuperate these lost stabilizing moments. Despite such design compensations, stable flight in flying wing and blended wing body designs can be difficult, due to reasons of limited moment arms reducing control effectiveness [89], strong coupling in the dynamics leading to nonlinear control effects [90], and the propensity for sudden pitch up behaviour at high angles of attack [91], a phenomenon known also as pitch-break.

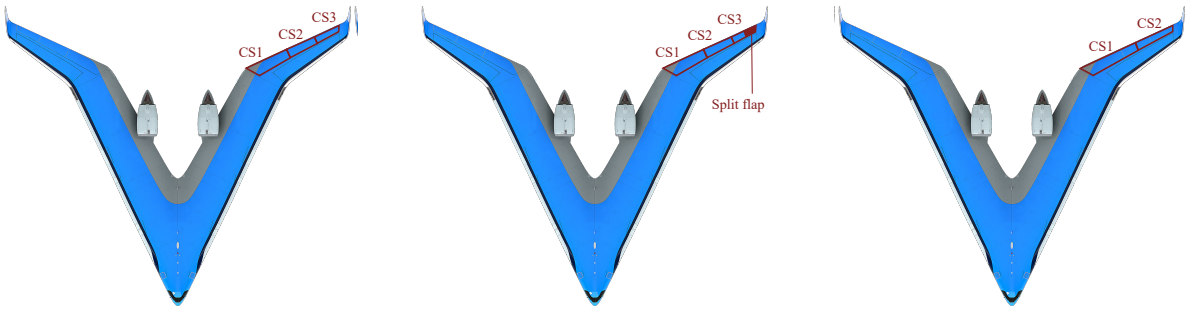
Regarding the flight dynamics of the Flying-V specifically, Cappuyns [92] performed numerical simulations of the design using a vortex lattice method, and observed that the handing qualities for some eigenmodes are slightly subpar, such as the Dutch roll mode during cruise, which is in fact slightly unstable as it has a negative damping factor. This observation was separately found by van Overeem et al. [93], who performed numerical studies of the Flying-V using mathematical models built from wind tunnel data, found that overall the handling qualities of the Flying-V were sub-par, and also observed that the Dutch roll mode was slightly unstable during cruise. During the approach, van Overeem et al. [93] saw that additional eigenmodes became unstable, namely the phugoid, spiral, and short-period modes. During low speeds, Torelli [94] found through piloted flight simulations of the Flying-V that it has a low damping

coefficient for its short period and phugoid eigenmodes. While Palermo and Vos [84] confirmed through wind tunnel experiments that the Flying-V also suffers from pitch break behaviours prevalent in highly swept-wing aircraft.

1.4.3. Flying-V Control System

For a flight controller, the most important design aspect of the Flying-V is the layout of its flight control system, specifically the control surfaces available and their control effectiveness. As the Flying-V does not have a tail like in traditional aircraft designs, the flight control surfaces that are traditionally located on the horizontal and vertical stabilizers of the empennage are moved to the main wing. This results in a control surface layout which features rudder surfaces on the vertical winglets providing control moments for yawing, and a series of control surfaces referred to as *elevons* along the trailing edge of the wing tip, which combines the functionalities of elevator and aileron providing control moments for rolling and pitching. Along the development history of the Flying-V, three main layouts for its flight control surfaces have been identified. While all three layouts used consistent rudder designs, they differed in how the elevons were configured:

1. Configuration adopted by Palermo [95], Palermo and Vos [84], and Garcia et al. [96] split the elevon into 3, an inner-board elevon CS1 used for pitch control, a mid-board elevon CS2 used for pitch and roll control, and an outboard elevon CS3 used for roll control.
2. Configuration adopted by Nolet [97] which also split the elevons into 3, but added a split flap surface out-board from CS3 elevons to provide additional yaw control and combat directional instability.
3. Configuration adopted by Cappuyns [92] split the elevons into 2, an inner-board elevon CS1 and an outer-board elevon CS2 used for pitch and roll control respectively.



(a) 3 elevons [84] **(b)** 3 elevons plus split flaps [97] **(c)** 2 elevons [93].

Figure 1.7: The various control surface setups used in various research efforts.

1.4.4. Dynamics Modelling

The core component of creating a flight control system is the aircraft dynamics model, without it, it is impossible to design let alone evaluate the performance of any controller. The same is true for reinforcement learning based controllers. To create such a model, experiments need to be performed to determine the aerodynamic stability and control coefficients of the Flying-V. This has been done in several ways, firstly through numerical Computational Fluid Dynamics (CFD) simulations of the full-scale aircraft, performed by Cappuyns [92] who used a Vortex Lattice Method (VLM). The second is through wind tunnel measurements, this was done in multiple campaigns using the 4.6% scale Flying-V model, one by Palermo and Vos [84] and another by Garcia et al. [96]. The third method is using flight test data, which was performed by a team from TU Delft in June 2021 [98].

The most up-to-date dynamics model of the Flying-V was developed by van Overrem et al. [93], which uses the **2 elevons configuration**, this model is built from two sources: the CFD based aerodynamic data and the wind tunnel experiments based data. This combination gives a model which is comprehensive when it comes to the dynamical properties of the Flying-V which are most interesting to capture,

specifically the unstable Dutch roll and lateral dynamics that the CFD data captures, and the pitch break behaviour present in the wind tunnel data but not the CFD or flight test data [98]. As this model is being used by several research projects running concurrently with this research, this model will thus also be used for the simulations in the present research.

1.5. 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.

1.5.1. Flight Control as an MDP

The goal of this thesis is to develop an intelligent flight control system for the Flying-V, 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 [99], the states **x** of the aircraft are shown in Equation 1.56, and the reference trajectory adopted defines targets for the sideslip, pitch, and roll states, as shown in Equation 1.57. 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 $\mathbf{x}_{ref}$ are first combined to produce an error vector **e**, defined in Equation 1.58. This error vector is then weighted producing $\mathbf{e}_w$, and concatenated with the aircraft's current control surface deflections and aircraft attitude rates, to produce the MDP states **s** shown in Equation 1.59. 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 $\mathbf{e}_w$, which is clipped to $[-1, 0]$, the norm of it taken and scaled, as shown in Equation 1.60.

$$\mathbf{x} = \begin{bmatrix} p & q & r & V & \alpha & \beta & \theta & \phi & \psi & h \end{bmatrix}^\top \tag{1.56}$$

$$\mathbf{x}_{ref} = \begin{bmatrix} \beta_{ref} & \theta_{ref} & \psi_{ref} \end{bmatrix}^\top \tag{1.57}$$

$$\mathbf{e} = \begin{bmatrix} \beta_{ref} - \beta & \theta_{ref} - \theta & \psi_{ref} - \psi \end{bmatrix}^\top \tag{1.58}$$

$$\mathbf{s} = \begin{bmatrix} \mathbf{e}_w^\top & \delta_a & \delta_e & \delta_r & p & q & r \end{bmatrix}^\top \tag{1.59}$$

$$r = -\frac{1}{3}\|\text{clip}(\mathbf{e}_w, -1, 0)\| \tag{1.60}$$

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 $\dot{\mathbf{x}}$ 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 1.61

$$\begin{aligned} \dot{\mathbf{x}} &= f(\mathbf{x}, \mathbf{u}) \\ \mathbf{y} &= g(\mathbf{x}, \mathbf{u}) \end{aligned} \tag{1.61}$$

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 $\mathbf{\Omega}$:

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

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:

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

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:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \end{aligned} \tag{1.62}$$

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 [100]. 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 1.61 obtained, defining the states and observations of the next time step:

$$\begin{aligned} \mathbf{x}_{t+1} &= f(\mathbf{x_t}, \mathbf{u_t}) \\ \mathbf{y}_t &= g(\mathbf{x_t}, \mathbf{u_t}) \end{aligned} \tag{1.63}$$

Finally, the deterministic form shown in Equation 1.63 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 [101].

1.5.2. 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 highly unstable [102]–[104] in order to perform fast manoeuvres for the least amount of input, but as a result can easily succumb to catastrophic loss of control. In contrast, commercial airliners are designed to be stable and easy to control [105], [106], 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. [107], 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. [108]managed to apply DDPG, specifically a benchmark implementation by Duan et al. [109][2], 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. [110] 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 1.8.

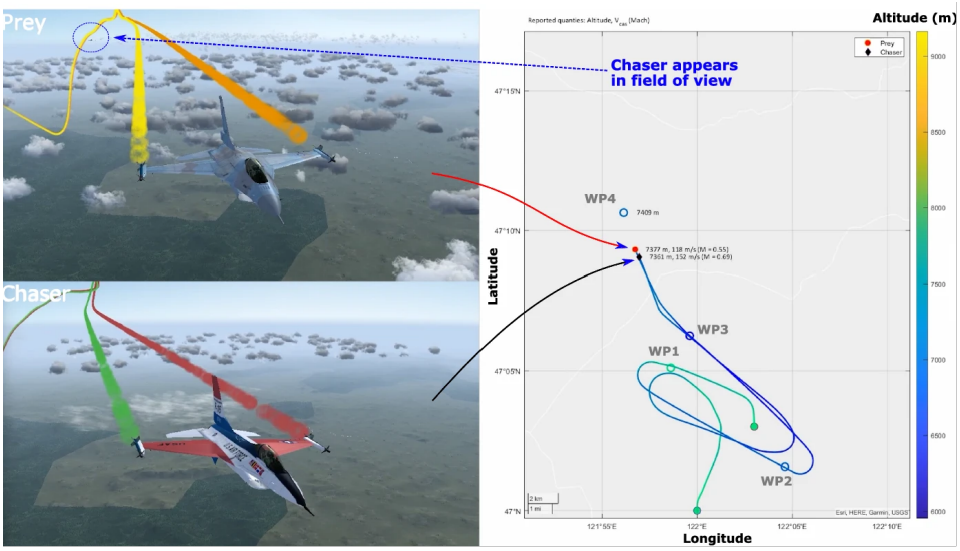


Figure 1.8: F-16 chaser simulation with DDPG pilots, screenshots of the jets in flight and their flight paths. Taken from [110]

This algorithm was also applied to the autonomous landing of fixed-wing aircraft in [111] and longitudinal control augmentation in [112].

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 1.5.1. 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

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

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. [113] 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 [99] show promise for continuing the use of SAC to the Flying-V. Interestingly, research from Zahmatkesh et al. [114] from some degree of fault tolerance can also be achieved when coupling a discrete Q-learning algorithm with fuzzy logic to 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 1.2.4, 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 [40].

Li et al. [115] 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. [40] 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 [116], 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. [41] 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 [117] 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 [118]. 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

When it comes to the Flying-V, there is an indubitable need for automatic flight control systems to be developed, firstly it's inherent handling qualities are not adequate for regulations, and secondly, there exists several eigenmodes where the aircraft is unstable, notably Dutch roll. Such characteristics are not the fault of poor aircraft design, but a matter of the difficult dynamics which a flying wing concept naturally presents. While previous work has investigated the matter of altitude tracking autopilots, there remains the question of how well can reinforcement learning based controllers handle attitude or rate tracking in the Flying-V. Additionally, it would be expected that further faults on the Flying-V that hinder it's lateral stability would be extra detrimental to the stable flight of this aircraft.

Furthermore, 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 [41]. Furthermore, there remain unexplored ideas for extending ACD algorithms. In early reinforcement learning algorithms such as the TD(λ) by Sutton [67], and in more recent ADP research [23], [24], [26], [119], [120], 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 [23], [24], or eligibility trace updates [119], [120]. 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.

1.6. 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.

In surveying 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 state-of-the-art 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 1.9. With these findings, **question 1a** is answered.

From the works of Dally and van Kampen, Casper and van Kampen, and various others, some 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, the various flight dynamic characteristics of the Flying-V that have been concluded in relevant literature sources show primarily that the lateral control issues as well as the lack of study on reinforcement based controllers focused on such dynamics, as elaborated on in Section 1.4 and under **Research Gap** in Section 1.5.2, help answer **question 2a, 2b**.

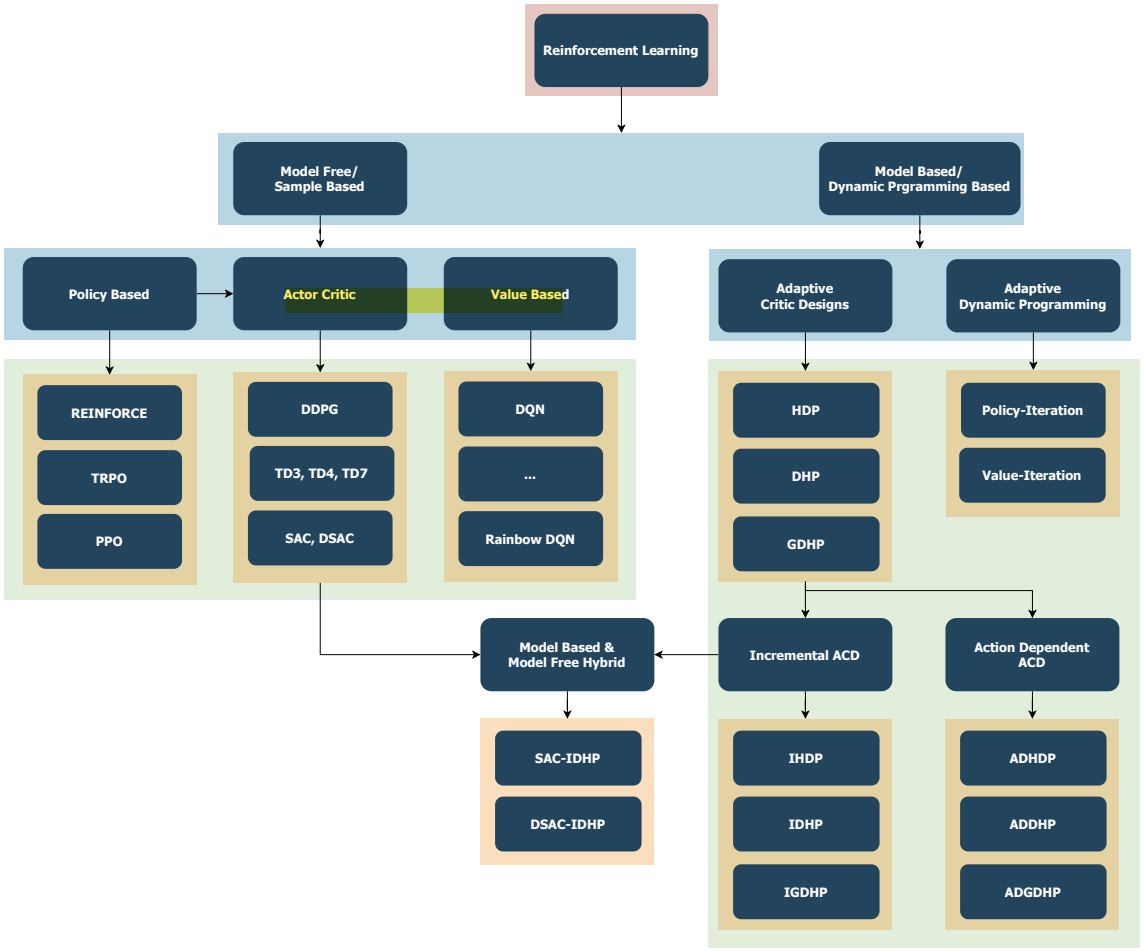


Figure 1.9: Overview of various reinforcement learning algorithms that have been encountered when compiling this literature study.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

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

[65] 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.

[66] 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.

[67] 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.

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

[69] 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.

[70] 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.

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

[72] 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.

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

[74] 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.

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

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

[77] 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.

[78] 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.

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

[80] J. Benad, “The flying v-a new aircraft configuration for commercial passenger transport,” in *Deutscher Luft-und Raumfahrtkongress 2015, Rostock*, 2015.

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

[82] M. Claeys, “Flying v and reference aircraft structural analysis and mass comparison,” 2018.

[83] L. van der Schaft, “Development, model generation and analysis of a flying v structure concept,” 2017.

[84] 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.

[85] B. Rubio Pascual and R. Vos, “The effect of engine location on the aerodynamic efficiency of a flying-v aircraft,” in *AIAA Scitech 2020 Forum*, 2020, p. 1954.

[86] W. Oosterom and R. Vos, “Conceptual design of a flying-v aircraft family,” in *AIAA AVIATION 2022 Forum*, 2022, p. 3200.

[87] W. Oosterom and R. Vos, “Conceptual design of a flying-v aircraft family,” in *AIAA AVIATION 2022 Forum*, 2022, p. 3200.

[88] N. van Luijk and R. Vos, “Constrained aerodynamic shape optimisation of the flying v outer wing,” in *AIAA AVIATION 2023 Forum*, 2023, p. 3250.

[89] C. Zhenli, M. Zhang, C. Yingchun, S. Weimin, T. Zhaoguang, L. Dong, and B. Zhang, “Assessment on critical technologies for conceptual design of blended-wing-body civil aircraft,” *Chinese Journal of Aeronautics*, vol. 32, no. 8, pp. 1797–1827, 2019.

[90] Y. Shan, S. Wang, A. Konvisarova, and Y. Hu, “Attitude control of flying wing uav based on advanced adrc,” in *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, vol. 677, 2019, p. 052 075.

[91] M. Tomac and G. Stenfelt, “Predictions of stability and control for a flying wing,” *Aerospace Science and Technology*, vol. 39, pp. 179–186, 2014.

[92] T. Cappuyns, “Handling qualities of a flying v configuration,” 2019.

[93] S. van Overeem, X. Wang, and E.-J. Van Kampen, “Modelling and handling quality assessment of the flying-v aircraft,” in *AIAA Scitech 2022 Forum*, 2022, p. 1429.

[94] R. Torreli, “Piloted simulator evaluation of low speed handling qualities of the flying-v,” 2022.

[95] M. Palermo, “The longitudinal static stability and control characteristics of a flying v scaled model: An experimental and numerical investigation,” 2019.

[96] A. Ruiz Garcia, R. Vos, and C. de Visser, “Aerodynamic model identification of the flying v from wind tunnel data,” in *AIAA Aviation 2020 Forum*, 2020, p. 2739.

[97] S. Nolet, “Improving the flying v directional control power by the implementation of split flaps,” 2022.

[98] K. Siemonsma, “Aerodynamic model identification of the flying-v using flight data,” 2022.

[99] 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.

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

[101] 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.

[102] 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.

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

[104] 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.

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

[106] 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.

[107] 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.

[108] 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.

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

[110] 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.

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] 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.

[120] 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.