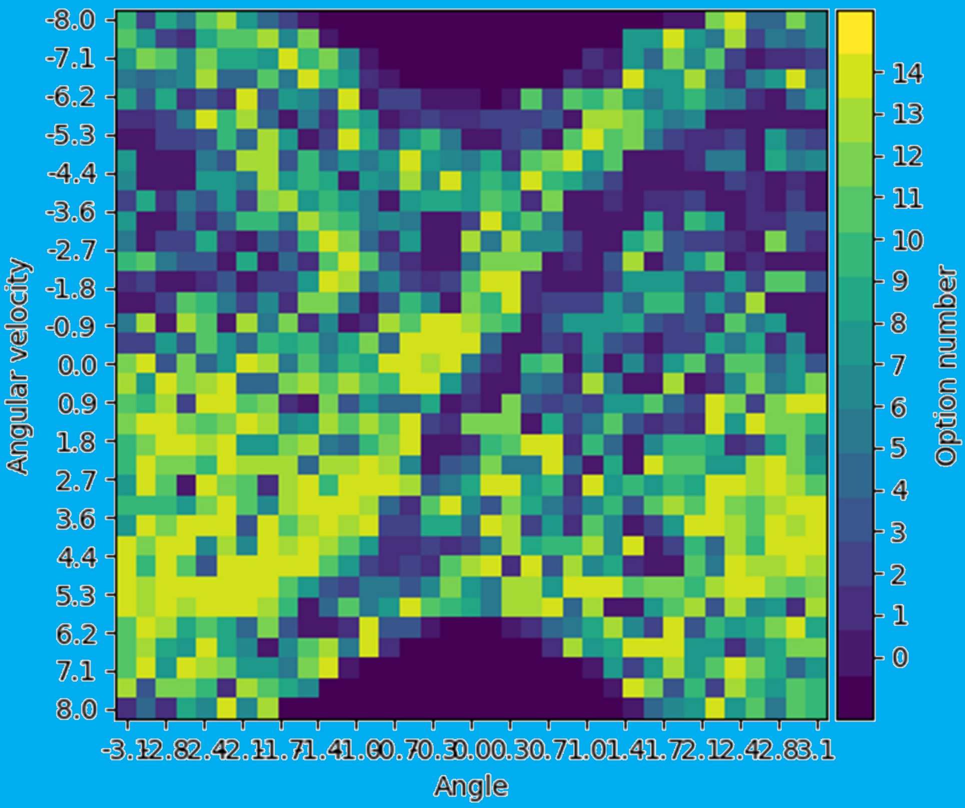
Hierarchical Reinforcement Learning for Model-Free Flight Control

A sample efficient tabular approach using Q(λ)-learning and options in a traditional flight control structure

J.M. Hoogvliet

Technische Universiteit Delft



TU Delft

Hierarchical Reinforcement Learning for Model-Free Flight Control

A sample efficient tabular approach using Q(λ)-learning and options in a traditional flight control structure

by

J.M. Hoogvliet

Delft University of Technology

Student number:	4135350	
Project duration:	March 19, 2018 – November 4, 2019	
Thesis committee:	Chair	Dr. ir. Q.P. Chu
	Daily Supervisor	Dr. ir. E. van Kampen
	Examiner	Dr. ir. B.F. Santos

Preface

To achieve my master's degree in Aerospace Engineering at the Technical University of Delft marks the end of my years as a student. Short hereafter I may call myself an Aerospace Engineer rather than proclaiming to just study it. I have enjoyed my time at this University and am curious to find out what is ahead. My final contribution to is this thesis report of which I am proud to have completed. At the start of this project I practically started out blank. Learning about reinforcement learning was enjoyable and had quite a bit more elements to it than I had anticipated.

Along my journey to complete this report I have had much help from the people around me. Firstly I would like to thank my supervisor Dr. ir. E.J. van Kampen with the support and open talks we had about many ideas. I remember in hindsight being a bit too enthusiastic about what could be accomplished with RL in the beginning. Later as my understanding developed we could converse about several ins and outs of the project, which was always very helpful. Also midway through the project when my progress slowed down he was always calm and gave me room. On the subject of perseverance I have to especially thank my girlfriend Mireille for her loving support and encouragement when I did get stuck. Sustaining the conversations about my endless thoughts really helped me though and this work would not have been possible without her.

Finally I have to thank my friends of 2.60 for giving me the necessary laughs, mountainbiking trips, and occasional gatherings. And my parents for being supportive along the way and proud of what I have achieved.

J.M. Hoogvliet
Delft, October 2019

Abbreviations

ADP	Approximate Dynamic Programming
AFCS	Automatic Flight Control System
CAS	Control Augmentation System
DHP	Dual Heuristic Programming
DoF	Degree of Freedom
DP	Dynamic Programming
DPG	Deterministic Policy Gradient
DDPG	Deep Deterministic Policy Gradient
DRL	Deep Reinforcement Learning
DQN	Deep Q-Network
FBW	Fly-by-wire
FA	Function Approximation
FCS	Flight Control System
FRL	Flat Reinforcement Learning
GPI	Generalised Policy Iteration
HRL	Hierarchical Reinforcement Learning
HAM	Hierarchies of Abstract Machines
IC	Intelligent Control
LTI	Linear Time-Invariant
MC	Monte Carlo
MDP	Markov Decision Process
MIMO	Multi-Input Multi-Output
ML	Machine Learning
PID	Proportional-Integral-Derivative
RL	Reinforcement Learning
SAS	Stability Augmentation System
SCAS	Stability and Control Augmentation System
SISO	Single-Input Single-Output
SMDP	Semi-Markov Decision Process
TD	Temporal Difference
UAV	Unmanned Aerial Vehicle

List of Symbols

Aircraft

α	Angle of attack
a_{nx}, a_{ny}, a_{nz}	Normal acceleration body frame, longitudinal, lateral and vertical
β	Sideslip angle
$\delta_a, \delta_e, \delta_r$	Aileron, elevator and rudder deflection angle
ϕ, θ, ψ	Roll, pitch, yaw angle
h	Altitude, Earth frame
M	Mach number
n_{pos}, e_{pos}	North and East position, Earth frame
n_z	Load factor
p, q, r	Roll, pitch, yaw rate
P_s	Static pressure
$\bar{q}$	Dynamic pressure
τ	Throttle set-point
u, v, w	Velocity vector body frame, longitudinal, lateral, vertical
V_t	Total velocity vector

Reinforcement learning

α	Learning rate
ϵ	Randomness factor
γ	Discount rate
G_t	Sum of discounted rewards
I, π, β	*Options*: Initiation set, policy and termination conditions
π	Policy
$Q(s, a)$	Q-table with Q-values of states [s] and actions [a]
r	Reward
$s, a, \mathscr{S}, \mathscr{A}$	State and action, state and action space
V	State-value

Misc.

θ	Angle
$\dot{\theta}$	Angular velocity
T	Torque

List of Figures

List of Tables

Contents

1 Introduction

Many small corrections are applied in-flight to an aircraft to remain in a stable state. Simultaneously larger motions can be executed to comply with the planned flight trajectory. On the modern flight deck these corrections and motions are generated by the on-board flight control system. At the heart of the flight control system are the control loops that link sensory information and control inputs to a control surface deflection. Designing a controller for such a system is traditionally done using a system model. With it, controllers are tuned to a variety of operating conditions using e.g. gain scheduling and state machines. The trend towards autonomy of vehicles puts more emphasis on control systems that are able to adapt to changes and uncertainties in the dynamics and can handle unforeseen circumstances. This requires a certain level of intelligence, an aspect that can be accomplished by adding the element of learning.

Reinforcement Learning (RL) is a model-free adaptive method for solving a stochastic optimal control problem formulated as a Markov Decision Process (MDP)[1]. It defines an agent, the controller, interacting with an environment, the aircraft. By observing a numerical reward signal the agent can learn how to behave such as to maximize this reward [1]. In regular flat-RL (FRL), the state-space of the learning problem is seen as a flat search space [2]. States and actions are enumerated and discretised into a tabular form. The size of the table grows exponentially with the number of states, known as the curse of dimensionality [3]. The number of samples required to learn a policy therefore also grows rapidly. The efficiency of each experience sample needs to be maximised to solve the problem within reasonable time and computation constraints.

This work uses Hierarchical Reinforcement Learning (HRL) to improve the sample efficiency of RL. HRL extends the framework of RL in two ways. Firstly it defines a learning structure that is a decomposition of the original problem. Each problem is of lower dimension and less influenced by the curse of dimensionality. Together they solve the larger problem. Secondly it uses the theory of semi Markov Decision Processes (SMDP) to add temporal abstraction of actions [4]. Instead of requiring decisions at each instant in time, temporally-extended activities can be triggered following their own policy until termination [5]. Using only decision points between activities reduces the required number of samples and guides the agent more surely through different parts of the search space. *Options* by Sutton is used to provide the temporal abstraction.

An existing hierarchical flight control structure can serve as a blueprint for the hierarchical decomposed learning structure of HRL. An example is cascaded proportional-integral-derivative (PID) control, where a PID controller in the outer loop controls the set-point for a PID controller in the inner loop [6]. Another example is gain scheduling, where controller parameters are changed for various operating conditions [7]. Both of these structures resemble a form of hierarchy, and are thus comparable to HRL. In this research an altitude reference tracking task is used to evaluate the impact of HRL. A traditional altitude reference tracking controller defines a three-level hierarchical controller structure for HRL.

The main objective is to investigate how HRL can be implemented to flight control, and how its performance compares to FRL. To facilitate this, a validated non-linear F-16 aircraft model is used [8]. Following the objective, this research aims to answer the main research question:

"How can the sample efficiency of a reinforcement learning based flight controller for a fixed-wing aircraft be improved with hierarchical reinforcement learning by using a learning structure analogous to hierarchies in existing flight control methods?"

From the main question several sub-questions can be formulated. Similar to hierarchical reinforcement learning these divide the larger problem into smaller sub-problems. Answering them will answer the main question in a step-wise manner:

1. How are flight control methods currently implemented?
    - How is hierarchy present in classical flight control?
2. How can reinforcement learning be applied to flight control?
    - What is state-of-the-art in flight control using reinforcement learning?
    - What is state-of-the-art in flight control using hierarchical reinforcement learning?
    - How can the hierarchical structure of existing flight control methods be converted into the framework of hierarchical reinforcement learning?
    - How should a reward signal be shaped?
3. What is the performance of the hierarchical reinforcement learning controller with and without temporal abstraction in relation to a controller based on flat reinforcement learning?
    - What is the number of samples required to learn an altitude reference tracking controller using FRL and HRL?
    - What is the tracking performance of the HRL controller in relation to FRL?
    - What is the sample efficiency of the HRL controller in relation to flat reinforcement learning?

This report is split into two parts. Part I is the scientific paper, the main body of the report. Part II concerns a review of literature, separated into flight control, reinforcement learning, hierarchical reinforcement learning, and the state-of-the art. Then, a preliminary analysis using an inverted pendulum is performed. Finally conclusions and recommendations about the project are given.

I

Scientific Article

Hierarchical Reinforcement Learning for Model-Free Flight Control

J.M. Hoogvliet*
Delft University of Technology, Delft, The Netherlands

Reinforcement learning (RL) is a model-free adaptive approach to learn a non-linear control law for flight control. However, for flat-RL (FRL) the size of the search space grows exponentially with the number of states, resulting in low sample efficiency. This research aims to improve the efficiency with Hierarchical Reinforcement Learning (HRL). Performance in terms of the number of samples and the mean tracking error is evaluated on an altitude reference tracking task using a simulated F16 aircraft model. FRL is used as the baseline performance index. HRL is used to define a three-level learning structure, re-using an existing flight control structure. Finally, *options* is used with HRL to add temporal abstraction. It is shown that by re-using the flight control structure the learning process is made more sample efficient. Adding *options* further increases this efficiency, but does not lead to better tracking performance.

Nomenclature

RL	=	Reinforcement Learning
FRL	=	Flat Reinforcement Learning
HRL	=	Hierarchical Reinforcement Learning
α, ϵ, γ	=	learning rate, randomness factor, discount factor
δ_t, e_t	=	temporal difference, eligibility trace
$\mathcal{I}, \pi, \beta$	=	*options*: initiation set, policy and termination conditions
$\beta_S, \beta_T, \beta_B$	=	*options*: termination conditions for state change, time and action bounds
k	=	discrete time-step
$\lambda_\alpha, \lambda_\epsilon, \lambda_e$	=	decay values for learning rate, greediness and eligibility
n_d	=	number of discretisations
π	=	policy
$Q(s, a)$	=	Q-table with Q-values of states [s] and actions [a]
r, G	=	reward, total reward in episode
$s, a, \mathcal{S}, \mathcal{A}$	=	state, action, state and action spaces
t, dt	=	time vector and time step
$\mathbf{u}, \mathbf{x}, \mathbf{y}$	=	system input, state and output vectors
v, N	=	number of visits and number of samples
α	=	angle of attack
δ_e	=	elevator deflection angle
$\gamma, \dot{\gamma}$	=	flight path angle, flight path angular speed
h	=	altitude
θ, q	=	pitch angle, pitch rate
T	=	thrust
V	=	velocity
ω_n	=	eigenfrequency

*MSc. Student, Control and Operations, Aerospace Engineering, Delft University of Technology

I. Introduction

Automatic flight control is traditionally achieved with a model of the aircraft. Techniques such as gain scheduling and state machines are used to design controllers for a variety of operating conditions. The trend towards increasing autonomy of vehicles requires control systems that are able to adapt to changes and uncertainties in the dynamics and can handle unforeseen circumstances. Model-free controller design is not limited to the used model and can therefore find control solutions different to those of traditional model-based design. An adaptive controller can improve performance over time and adapt to changes of the aircraft. Both properties are important to increase aircraft autonomy in the future.

Reinforcement Learning (RL) is an adaptive, model-free method for solving a stochastic optimal control problem formulated as a Markov Decision Process (MDP) [1]. The MDP defines interaction between an agent and the environment, where it is the goal of the agent to learn to maximize the sum of rewards it receives over time in response to its selected actions. It does so by collecting experience samples $< s, a, r, s' >$ consisting of state, action, reward and next state. The nominal form of RL, 'flat'-RL as mentioned by Dietterich [2], enumerates and discretises each state and action into a tabular form, formulating it as a flat search space. An issue of using this type of RL is the combinatorial explosion of the table size with the number of states, known as the curse of dimensionality [3]. For higher dimensional problems this is why flat-RL is impractical, infeasible or too sample inefficient to solve within reasonable computation- and time-constraints. The efficiency of each experience sample needs to be maximized to bring down the expenses required.

The aim of this study is to improve the sample efficiency of flat-RL with Hierarchical Reinforcement Learning (HRL) by re-using the structure of existing traditional flight control logic for its hierarchical learning structure. To facilitate learning a validated F-16 aircraft model [4] is used as the environment for the agent to interact with. Although strictly speaking this is a model, the agent has no knowledge of its inner workings. It is merely a way of generating experience samples. An altitude reference tracking task is used as the test-bed. Performance on this task is assessed in terms of the mean tracking error and sample efficiency.

HRL is a twofold extension of RL. Firstly it acknowledges the possibility of a hierarchical decomposition of a problem. Smaller dimensional problems together solve the larger problem whilst each individual lower dimensional piece suffers less from the curse of dimensionality and thus require less samples to learn. Secondly, it introduces temporal abstraction, the notion that actions have a duration. Instead of requiring decisions at each instant in time, temporally-extended activities can be triggered following their policy until termination [5]. By only using samples of decision points between activities the sample efficiency is further increased. *Options* by Sutton et al. [6] is used to provide the temporal abstractions.

To evaluate differences in sample efficiency, three control strategies are implemented. The first is flat-RL, used as the baseline score. The second is HRL without temporal abstraction, defining a three-level learning structure. Finally HRL with *options* is used to add temporal abstraction of actions. Model-free learning is implemented using Q-learning, a well known form of RL by Watkins [7]. Q-learning has good convergence guarantees and is easily implemented [8]. For both FRL and HRL Q-learning is used in combination with replacing eligibility traces, Q(λ) [7, 9]. Eligibility traces offer significant improvements in learning speed and reliability, and help in dealing with delayed rewards [10]. *Options* is used without traces.

Reinforcement learning in general has been used on several occasions for achieving flight control. The majority of which is applied on Unmanned Aerial Vehicles (UAV) for real world experiments. A common approach is a two-phase learning procedure either by using offline learning or expert demonstrations as the first phase, and thereafter improve performance online. This technique has shown successful results for the control of an autonomous helicopter using policy-based RL, where Bagnell and Schneider [11] learned a robust model-based helicopter flight controller, Ng et al. [12] learned autonomous inverted helicopter flight, Abbeel et al. [13] demonstrated more advanced helicopter aerobatics, and Coates et al. [14] have shown that by learning from expert demonstrations the helicopter can outperform the previous state of the art and the expert itself. Value-based methods have been less popular in flight control. Ferrari and Stengel [15] used an adaptive-critic approximate dynamic programming approach to successfully control a six degree of freedom jet over its full operating range. Q-learning was used by Molenkamp et al. [16] to autonomously switch activation of the best pre-made controller for aggressive quad-copter manoeuvres, and by de Vries and van Kampen [17] to control altitude for an over-actuated aircraft. HRL, however, is a scarcity in flight control. Recently, Mannucci et al. [18] showed the use of HRL in the field of safe reinforcement learning, improving exploration when crashing is not desirable. Verbist et al. [19] demonstrates this in learning a two-dimensional quad-copter control law, where HRL was more effective than flat-RL in learning and in safety of exploration.

This paper contributes to the development of hierarchical reinforcement learning for flight control. The current presence of HRL in flight control is small. Contrary to traditionally based methods and most applications of RL to flight control, this research takes a model-free approach. Although use is made of kinematic relations between state variables in the form of a flight control structure, this merely serves as a framework in which learning takes place. It does not require modelling of the aircraft. A final contribution is the use of a value-based method (Q-learning), where most approaches make use of policy-based methods or actor-critic architectures. The emerged three-layer controller can be analysed for its intermediate control responses, making its output generation process more transparent than a flat approach. Its model-free adaptive nature sets a step in the direction of autonomy for future aircraft control systems.

The remainder of this paper is divided in six sections. Section II gives an overview of RL and HRL and the specific methods used. Section III provides an description of the used simulation model. Next, Section IV introduces the three control strategies that are compared in this research. In Section V the setup of the conducted experiment is presented and in Section VI the results are evaluated and discussed. Finally in Section VII conclusions are drawn and recommendations for future research are given.

II. Reinforcement Learning

Reinforcement learning (RL) is a model-free method for solving problems formulated as a Markov Decision Process (MDP). Such problems are described by the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$, a set of states, set of actions, a state transition function and a reward function. A MDP defines an interface between an agent and the environment. For control purposes the agent embodies a controller, and the environment expresses the dynamics of a system. The agent is free to explore the environment by selecting and executing control actions. It then observes the next state of the environment and a corresponding numerical reward signal [1]. In doing so, the agent collects experience samples $< s, a, r, s' >$ consisting of the state, action, reward and next state. It is the goal of the agent to use these experience samples to improve its action selection process and so maximize the sum of rewards over time. By the idea of generalised policy iteration (GPI), the agent will converge to an optimal policy π^* [1].

A. Q(λ)-learning

A well-known technique in RL is Q-learning [7]. At its core, an action-value function represented by a table of Q-values is maintained for each state-action pair. These values represent the quality of each action from a state as the expected total discounted reward of taking action a and following policy π thereafter. Q-learning is easily implemented and provides good convergence guarantees [8]. The Q-table is updated with experience samples $< s, a, r, s' >$ gathered by interacting with the environment. This update is formulated as an in-place modification of the current Q-value, shown in Eq. 1, where γ is the discount rate and α the learning rate or step-size parameter. These parameters ought not to be confused with the aircraft state variables for flight path angle γ and angle of attack α respectively.

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(s,a)\left[r + \gamma \max_{a \in \mathcal{A}} Q_t(s',a) - Q_t(s,a)\right] e_t(s) \tag{1}$$

Trailing parameter $e_t(s)$ is the eligibility value of the to-be-updated state s. Q-learning belongs to the family of temporal difference (TD) methods within RL. Contrary to Monte-Carlo (MC) methods which require an episode to be completed before the agent is updated, TD methods will update after each time-step. The addition of eligibility traces (λ) essentially forms a hybrid between TD and MC, updating multiple visited states eligible for an update at once. This unifies and generalises the TD and MC methods [1]. Eligibility traces in general offer significant improvements in learning speed and reliability, and are a basic measure for dealing with delayed rewards [10]. Given a good step-size α is set, replacing traces provide more stability over accumulating traces [10]. The eligibility value is updated each time-step as shown in Eq. 2 by decaying it with $\gamma\lambda_e$ over time, except for the value of the current state which is replaced by one. The collected trace of visited states is reset upon selection of an exploratory action, i.e. in random action selection.

$$e_{t+1}(s) = \begin{cases} \gamma\lambda_e e_t(s) & \text{if } s \neq s_t; \\ 1 & \text{if } s = s_t; \end{cases} \tag{2}$$

The Q-table is optimistically initialised. To extract a policy from the Q-table, a maximization over all non-zero Q-values for each state-action pair is performed with Eq. 3. These values correspond to unexplored or un-reached state-action combinations and contain no information. If none of the actions are explored for a state the first entry is selected by default, corresponding to the lowest action available. The result is a policy that specifies the action a to be taken in state s as a simple table-lookup operation. Because of the maximization over Q-values, the dimension of the policy is equal to the dimension of the Q-table minus one: $\pi^{dim} = Q(s,a)^{dim} - 1$.

$$\pi(a|s) = \max_{a \in \mathcal{A}} \{Q(s,a) \mid Q(s,a) \neq 0\} \quad \forall s \in \mathcal{S} \tag{3}$$

B. Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) extends twofold onto the framework of RL. Firstly it defines a decomposition of the problem into several smaller ones. Smaller problems are less influenced by the curse of dimensionality. By solving each smaller problem, the overall problem is solved. Secondly, HRL adds temporal abstraction, the notion that actions have duration. Instead of requiring a decision at each point in time, temporally-extended activities are initiated following their own policy until termination [5]. This abstraction is based on the theory of semi-Markov Decision Processes (SMDP), an extension of the MDP where actions take variable amounts of time [6]. The discrete-time SMDP formulation underlies most approaches to HRL [5]. The three most popular approaches are *options* [6], MAXQ [2] and HAMs [20]. They have shown that by introducing a hierarchy the sample efficiency can be improved significantly [2]. Q-learning also applies to learning under the HRL formulation, if the one-step reward is interpreted as the accumulated discounted return from the execution of a temporally-extended activity. Equation 4 shows Q-learning for the SMDP case, where τ is the time taken for an activity to complete [5]:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha \left[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \gamma^{\tau} \max_{a\in\mathcal{A}} Q_t(s',a) - Q_t(s,a) \right] \tag{4}$$

This work uses *options* by Sutton et al. [6] to implement temporal abstraction. *Options* is closest to the MDP formulation of flat-RL. The available set of policies at each decision point in the SMDP chain are known as *options*, a closed-loop control rule [21]. Each *option* consists of an initiation set, a policy and termination condition: $< I, \pi, \beta >$. The initiation set determines from which state the *option* can be started, thereafter following π. The termination conditions can contain anything that designates a target has been reached, or the policy is taken outside of its applicable domain [6]. In contrast to the MDP, this termination condition can include a period of elapsed time or be dependent on the history experienced by the agent. After termination an update to the Q-table is performed. Equation 5 shows how Q-learning is modified to accept the duration of an action τ such to learn a policy over *options*. If the *option* designates a one-step *option*, this rule is equal to the normal Q-learning update rule, a policy over primitive actions.

$$Q_{t+1}(s,o) \leftarrow Q_t(s,o) + \alpha \left[r + \gamma^{\tau} \max_{o\in O} Q_t(s',o) - Q_t(s,o) \right] \tag{5}$$

There are several advantages to the use of *options* for learning over flat-RL. The formulation of *options* is closest to action selection in flat-RL, so planning and learning as performed in flat-RL remains the same [5]. The use of *options* is also beneficial in the initial stages of learning, where *options* prevent a period of ‘flailing’ often seen in RL systems [5]. In addition, and similar to other hierarchical approaches, *options* facilitate the transfer of learning knowledge to similar tasks. But this depends entirely on the decomposition of the problem and the task of the *option*. The focus of *options* as a method is to augment the core MDP with temporal extensions [6]. However, if the set of *options* does not include the one-step *option* corresponding to the entire set of primitive actions, *options* simplify rather than augment [5]. In this work *options* have been used as augmentation only. Each hierarchical layer produces primitive actions that are temporally extended using the theory of *options*.

III. Aircraft Environment

The flight dynamics of the environment are provided by an F-16 aircraft model obtained from Russel [4]. It simulates the dynamics of the real aircraft shown in Fig. 1. Both a low- and high-fidelity model are available, respectively provided by Stevens and Lewis [22] and Nguyen et al. [23]. For this work the low fidelity model was chosen. The adaptive nature of RL makes it possible to use a simplified model offline and thereafter improve it online on the actual airframe. This second step is out-of-scope, and merely serves as reasoning behind the model choice.

The focus of this work is on the design of an altitude reference tracking controller in longitudinal space. To obtain the longitudinal equations of motion the system is trimmed and linearised about an altitude of 15000 ft and a velocity of 500 ft s^{-1}. The resulting state-space system and trim state are shown in Eq. 6 and Table 1 respectively. This information is used to build an environment class in Python. Its task is to provide RL functionality such as the generation of observations, transitioning to the next state and resetting before every new episode. It propagates the internal state $\mathbf{x}$ through time in response to input vector $\mathbf{u}$ with a Runge-Kutta4 (RK4) integration scheme at a sample rate of 100Hz.

$$\begin{aligned} \dot{\mathbf{x}} &= A\mathbf{x} + B\mathbf{u} \\ \mathbf{y} &= C\mathbf{x} + D\mathbf{u} \end{aligned} \tag{6}$$

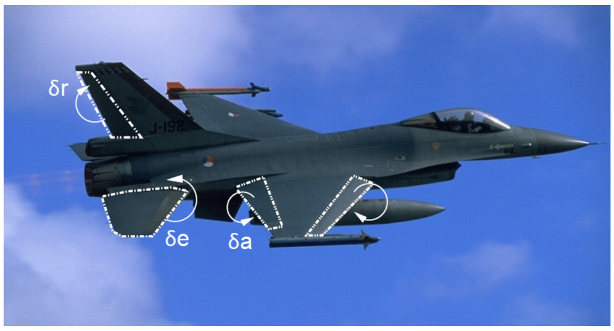


Fig. 1 F16 fighter jet impression, adapted from Russel [4]. Control surfaces and their positive deflections are indicated: elevator deflection δ_e, rudder deflection δ_r and aileron deflection δ_a.

To the longitudinal state-space matrices that return from the linearisation two additional states are augmented: the flightpath angle γ (= $\theta - \alpha$) and flightpath angular speed $\dot{\gamma}$. These are useful quantities for altitude reference control. The internal state vector is then represented by nine states, shown in Eq. 7. Output vector **y** is chosen to reflect the internal state, since all states can be of interest to the learning process. Problem-specific observation vectors for the agent are formed by selecting from **y**. Values of **x** and **y** are relative to the trim state of the aircraft. Table 1 lists these values, their unit and SI-unit.

$$\mathbf{x} = \mathbf{y} = \begin{bmatrix} h & \theta & V & \alpha & q & \gamma & \dot{\gamma} & T & \delta_e \end{bmatrix}^T \tag{7}$$

Input vector **u** is of size two in the longitudinal domain, comprising of the thrust and elevator deflection. Because of the presence of actuator dynamics, these quantities are also seen present in the internal state vector **x**. Conform RL convention, input **u** will be referred to as the action vector a_t.

$$\mathbf{u} = \mathbf{a}_t = \begin{bmatrix} T & \delta_e \end{bmatrix} \tag{8}$$

Table 1 Longitudinal aircraft trim state.

Variable	Value	Unit	SI Value	Unit	Description
h	15000.00	ft	500.00	m	Altitude
θ	0.08	rad	4.47	deg	Pitch
V	500.00	ft s^{-1}	152.40	m s^{-1}	Velocity
α	0.08	rad	4.47	deg	Angle of attack
q	0.00	rad s^{-1}	0.00	deg s^{-1}	Pitch rate
γ	0.00	rad	0.00	deg	Flight path angle
$\dot{\gamma}$	0.00	rad s^{-1}	0.00	deg s^{-1}	Flight path angular speed
T	2120.62	lbs	9432.99	N	Thrust
δ_e	-2.46	deg	-2.46	deg	Elevator deflection

The predominant longitudinal eigenmodes of the aircraft are the short period- and phugoid motion. With their respective eigenvalues known, the damping, eigenfrequency and time-period can be calculated, shown in Table 2. Both short period and phugoid modes are stable and have periods of 5.2 seconds and 76.3 seconds respectively. The eigenfrequencies and periods of motion are metrics used in the design of appropriate training signals for the agent.

IV. Control Design

Three different RL strategies are implemented to learn an altitude reference tracking controller. They are evaluated for their performance in terms of learning stability, mean tracking error and sample efficiency. Flat-RL is used as

Table 2 Eigenmodes of the longitudinal model

Eigenvalue	Mode	Damping ζ, -	Frequency ω_n, s^{-1}	Period T, s
$-0.7621 \pm 1.2051i$	Short period	0.5345	1.4259	5.2
$-0.0039 \pm 0.0843i$	Phugoid	0.0462	0.0844	76.3

a baseline design to determine the learning- and tracking performance in the nominal case, strategy **i**. Then two hierarchical strategies are developed, using a traditional three-layer altitude reference tracker. Control strategies **ii** and **iii** therefore both consists of three hierarchical layers. The first is built using HRL without temporal abstraction, and the second and final design uses HRL with *options*. An overview is given in Table 3.

Table 3 Different reinforcement learning strategies. Problem size indicated as a function of the number of discretisations (n_d) per state and action.

num.	abb.	Strategy	Problem size	Learning structure
i	FRL	Flat RL	$(n_d)^{S+\mathcal{A}}$	Flat Reinforcement Learning (baseline)
ii	HRL	Hierarchical RL	$3(n_d)^{S+\mathcal{A}}$	Hierarchy of FRL controllers
iii	HRL	Hierarchical RL + *options*	$3(n_d)^{S+\mathcal{A}}$	HRL, including temporal abstraction

A. Flight control schematic

Altitude reference tracking control requires a controller that minimizes the error between the reference height and the actual height. The most effective longitudinal input of the system to accomplish an altitude change is the elevator deflection δ_e. This parameter is chosen as the controlled input to the aircraft system, i.e. the action generated by the agent. Since the velocity of the aircraft changes as a result of deviations in altitude (potential and kinetic energy exchange), a velocity-hold controller is implemented separately using proportional control: $T = 300\Delta V$, with ΔV the deviation in ft s^{-1} from the trim value. The flight control structure used to steer the elevator deflection is a collaboration of three reference tracking controllers, shown in Fig. 2, connected to each other in an inner-outer loop structure, each generating a reference signal for the next-in-line.

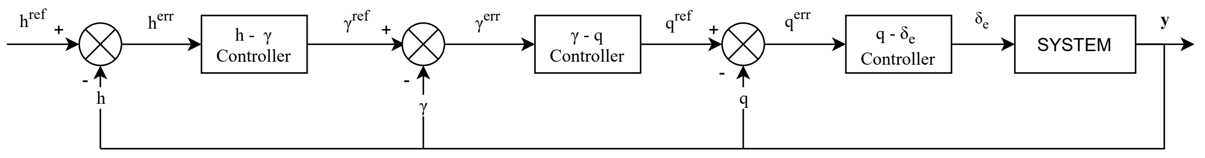


Fig. 2 Hierarchical decomposition of altitude reference control into three control layers.

The control variables of this flight control schematic are, in order: h$^{\text{ref}}$, γ^{ref}, q$^{\text{ref}}$ and δ_e. Before being fed directly to the next controller they are first compared to their actual value, obtaining the corresponding error signal. This introduces the controller input variables h^{err}, γ^{err} and q^{err}.

Each of the reference tracking controllers act on a different dynamic portion of the aircraft, having different time-scales. The time-scale here is understood as the relative length of time a controller needs to physically drive the system to track a reference. This difference in time-scale is one of the main reasons why the longitudinal dynamic relations are split into the hierarchical order of altitude - flightpath - pitch rate - elevator. It minimizes the influence of one hierarchical level onto the other. It is the same principle on which cascaded PID loops are designed, the fastest responding controller is put in the inner loop.

The eigenmotions from Table 2 are used to determine a time-scale value for each controller. The phugoid and short period eigenvalues are inverted to retrieve a time length. To be sure that also higher magnitudes can be tracked 50% margin is added to this value: $1.5/\omega_n$. The top level $h - \gamma$ controller time-scale is based on the phugoid value and equals 17.77 seconds. The lowest level $q - \delta_e$ controller time-scale is based on the short period eigenvalue and equals 1.05

seconds. The time-scale of the intermediate controller is taken as the average between them: 9.41 seconds. Time-scale and other problem-specific settings are found alongside the hyper-parameters in Table 5.

B. Flat Reinforcement Learning Control (i)

Simply using all of the available observation variables **y** defined in Eq. 7 as information for the agent would require a ten-dimensional table including the elevator deflection action variable. As shown in Fig. 2 the altitude h, flight path angle γ and pitch rate q suffice for altitude control. For flat-RL these states are therefore selected as the agent's state representation. The Q-table for this agent becomes an enumeration of combinations in four dimensions, one dimension for each state and action. Internal order and relations between variables as defined by the hierarchical decomposition of Fig. 2 are thereby lost and not known to the agent. Flat-RL tries to learn the correct elevator command directly from the value of these three state variables. Figure 3 gives a graphical representation of the four dimensional internal Q-table. As the table size a representation using 25 discrete values is chosen.

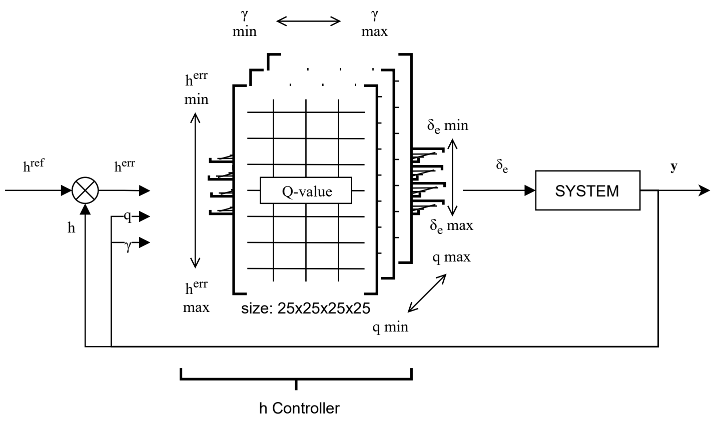


Fig. 3 Flat reinforcement learning controller diagram. A four-dimensional flat Q-table representation of the altitude reference tracking control problem.

Q-table inputs are shown as a combination of $h^{\text{err}}, q, \gamma$, defining Q-values for each δ_e. Only the altitude is formulated as an error signal, since it has a reference input. Both altitude and pitch rate tend to go to zero as the altitude reference is satisfied. Flight path angle γ does not necessarily follow this rule, and might be non-zero in case of e.g. a ramp signal.

C. Hierarchical Reinforcement Learning Control (ii + iii)

The hierarchical strategy splits the controller into three parts by the decomposition of Fig. 2. Each part or level generates a reference command for the next-in-line, effectively creating three separate layers of reference tracking control. The immediate gain of this decomposition is the reduction in overall problem size complexity. For flat-RL the table size at 25 discretisations per state and action is: 25^4. Splitting the problem into three equal parts results in a size of $3 \cdot 25^2$, a 208-fold decrease. Each level is namely described by a single state and a single action value, a two-dimensional table. Figure 4 indicates the data flow and table order.

For a discrete value of h^{err} the agent has to learn the appropriate Q-values for each γ^{ref} output. The next hierarchical layer has to decipher how to steer q^{ref} as response to a given γ^{err}, and so forth. Different to the flat-RL approach each variable is now defined as an error state, each output as a reference. Satisfying a command is done by regulating all errors to zero. If successful, the agent will therefore spend most of its experience samples around the zero-mark.

The schematic shown in Fig. 4 is a tabular representation of Fig. 2 and is the same for HRL with and without temporal abstraction. The only difference is the addition of *option* logic to facilitate the duration of an action. Three termination conditions are defined: β_S, β_T and β_B. The first defines termination upon a change of state, where the current state is not equal to the state the *option* was initiated from. This forces decisions to be made for each state. The second termination β_T, defines a maximum number of time-steps before termination, and finally β_B terminates the *option* if the given reference action command is satisfied within given bounds B. The duration of actions are therefore bounded by β_T, but can be as quick as the underlying controller is able to satisfy its generated reference command. There are no restrictions set on $\mathcal{I}$; *options* can be initiated from every state in $\mathcal{S}$.

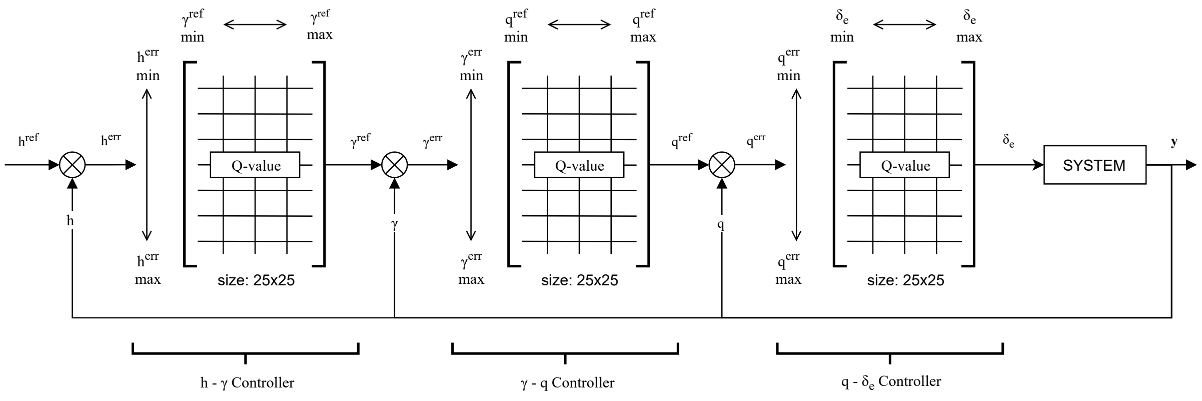


Fig. 4 Hierarchical reinforcement learning controller diagram. Three two-dimensional representations of the altitude reference tracking control problem.

Because *options* do not make use of eligibility traces, the computational complexity with respect to HRL is lower. The maximum length of a trace whilst using $\lambda_e = 0.9$ is 30. Every update consists therefore of 30 Q-table updates. Without *options* updates also happen at every time-step. The reduction in computational complexity for options can therefore be as much as 600 for a temporal abstraction of 20 simulation steps. This property is especially interesting for online performance, where computational expenses might be scarce.

V. Experiment

The experiment concerns the evaluation of the reference tracking performance for FRL, HRL and HRL with temporal abstraction. To train each controller, reference signals are generated randomly on an episodic basis. To evaluate the policy, greedy action selection is performed on four test signals, determining the average tracking error on those signals. Both training- and test reference signals are matched to the time-scale and discretisation input range of the controller. Each controller therefore is trained according to its physical capabilities. Data from learning is evaluated in terms of policy variance and number of samples. After training of FRL and the individual elements of HRL and assembly into a controller, the control methods are compared on a set of six altitude reference tracking tasks.

A. Discretisation settings

For Q-learning, each variable is discretised to assign its value to an entry in the Q-table. The parameters used throughout this experiment are the same for each of the controller strategies; altitude h, pitch rate q, flight path angle γ and elevator deflection δ_e. Each is split into 25 bins, where the odd number is chosen deliberately to include the zero-error state. Boundaries for the elevator deflection are constrained to 12.4 degrees either side, giving plenty of control authority with the F16 aircraft. Pitch rate is set to 10 degrees per second and the flight path angle is discretised between -20 and 20 degrees. The altitude falls within 250 meters. For sake of simplicity the reference output and error input discretisation domains are matched. Note however that it is also possible to have a command signal that is larger than the error domain defines. Potentially this can increase peak performance whilst retaining the same number of discretisations. Table 4 presents an overview.

Table 4 Discretisation settings per parameter

Parameter	n_d	domain	unit
h^{ref}, h^{err}	25	[-250, 250]	m
$\gamma^{ref}, \gamma^{err}$	25	[-20, 20]	deg
q^{ref}, q^{err}	25	[-10, 10]	deg s^{-1}
δ_e	25	[-12.4, 12.4]	deg

1. Discretisation scaling

During learning some states may be visited more often than others. Especially in the case of error minimisation, most samples are spend around the zero-mark in the centre of the discretisation domain. As a result, the state-visit dependent variables α and ϵ can be vastly different from their surrounding states. This can result in worse performance because the discretisation domain is used less effectively. To gain finer control over the area the agent spends most time, as well as to smooth out the experience samples over a larger part of the search space, thereby improving exploration and gradual learning, the error state discretisation domains are scaled towards the centre. A visualisation of the scaling is shown in Fig. 5, where the left figure shows normal discretisation, and the right figure scaled intervals. The effect of scaling on the state visit counts per state is shown in Fig. 6, where the distribution of collected experience samples for the same number of samples are shown for q^{err} of controller $q - \delta_e$. Whenever a variable is not located within the discretisation domain, it is assigned to the edge bin. These values therefore show up as more frequently sampled by the agent in both distributions.

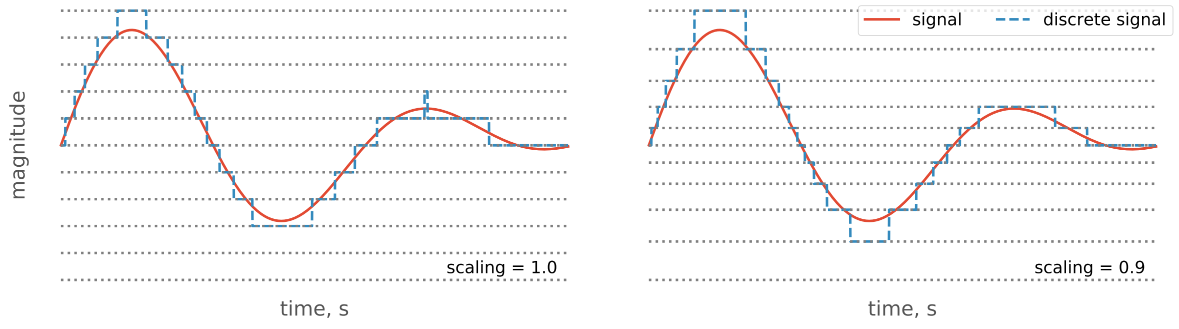


Fig. 5 Discretisation scaling visualised. Uniform distribution in the left figure, scaled discretisation in the right with factor d=0.9.

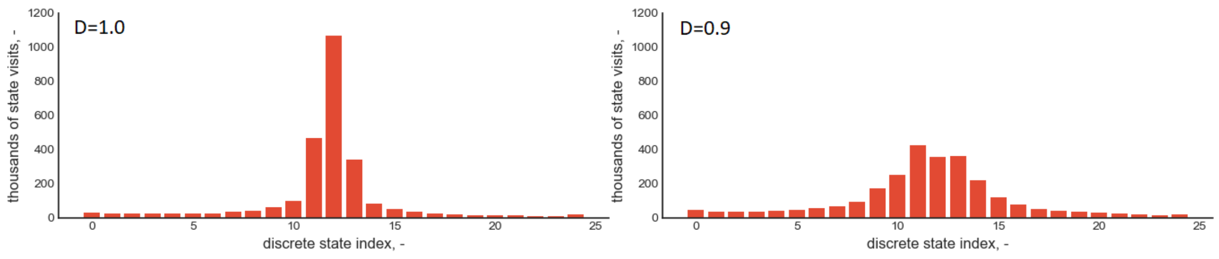


Fig. 6 A comparison of q^{err} state visit counts for controller $q - \delta_e$ between a discretisation scaling of 1.0 and 0.9 for the same number of experience samples N.

B. Training Procedure and Set-up

The goal of the training procedure is to provide learning stability, and to converge to an optimal policy. Because randomness is an inherent property of RL, the policy resulting from the same training can be different between trails. Without stable learning this difference can vary greatly between multiple training attempts. As a consequence it is meaningless to comment on the control stability and reference tracking performance of these policies when there are no guarantees that learning converges to roughly the same policy each time. Variance between policies should therefore be minimized. A second goal to the training procedure is to reach the optimal policy with high sample efficiency. This corresponds to fewer samples used for the same tracking performance.

1. Agent set-up and hyper-parameters

The agent is optimistically initialised, and learning is commenced in an episodic manner. Episode lengths differ per hierarchical layer to fit the reference training signals and are 20 seconds, 40 seconds and 60 seconds respectively from lowest to highest hierarchical layer. The eligibility decay factor λ_e is set to 0.9 for all methods without *options*. *Options* is run without eligibility traces. For action-selection an ϵ-greedy strategy is used, and to encourage exploration in the initial phase of learning, both ϵ and step-size α are made to decay exponentially with the number of state-action visits using Eq. 9. The optimal decay value varies per problem, and is therefore subject to tuning. The number of visits that drive the decay are incremented at every Q-table update with the current eligibility value: $v(s,a) = v(s,a) + e_t$. The number of visits therefore represents the total contribution factor of each state-action combination. For greediness updates, $v(s)$ is taken as the mean visits over actions in $v(s,a)$.

$$\alpha = \alpha_0 \left(\frac{1}{2}\right)^{v(s,a)\lambda_\alpha} \qquad \epsilon = \epsilon_0 \left(\frac{1}{2}\right)^{v(s)\lambda_\epsilon} \tag{9}$$

To assure convergence with probability 1, the choice of the step-size α and its variation over time must adhere to two general stochastic approximation conditions [1]: $\sum_{n=1}^{\infty} \alpha_n(a) = \infty$ and $\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$. Large enough to overcome initial conditions or fluctuations and eventually small enough to assure convergence. The step-size is set to decay to a minimum of 0.01, while ϵ decays to zero. Table 5 shows the hyper-parameter settings and signal properties per problem. Two entries for FRL exist, this will be explained in the results section. The right half of Table 5 contains information about the training signal magnitudes, the time-scale of each problem and the reward signal as the negative absolute value of the state error. The left half of the table shows the agents' learning settings. These have been tuned to produce decent policies.

Table 5 Hyper-parameters and signal settings for each control strategy.

Strategy		α	λ_α	ϵ	λ_ϵ	β_B	timescale	ref magnitude	r_t
FRL (25^4)		[0.5, 0.05]	0.001	[0.1, 0.01]	0.004	-	17.77 s	[-250, 250] m	$-\lvert h^{err}\rvert$
FRL ($9 \cdot 7 \cdot 7 \cdot 11$)		[1, 0.01]	0.001	[0.5, 0]	0.004	-	17.77 s	[-250, 250] m	$-\lvert h^{err}\rvert$
HRL	$h-\gamma$	[1, 0.01]	0.004	[0.5, 0]	0.004	-	17.77 s	[-250, 250] m	$-\lvert h^{err}\rvert$
	$\gamma-q$	[1, 0.01]	0.001	[0.5, 0]	0.001	-	9.41 s	[-20, 20] deg/s	$-\lvert \gamma^{err}\rvert$
	$q-\delta_e$	[1, 0.01]	0.001	[1, 0]	0.0005	-	1.05 s	[-10, 10] deg	$-\lvert q^{err}\rvert$
HRL	$h-\gamma$	[1, 0.01]	0.004	[0.5, 0]	0.0067	0.25 deg	17.77 s	[-250, 250] m	$-\lvert h^{err}\rvert$
+*options*	$\gamma-q$	[1, 0.01]	0.004	[0.5, 0]	0.008	0.25 deg/s	9.41 s	[-20, 20] deg/s	$-\lvert \gamma^{err}\rvert$
	$q-\delta_e$	[1, 0.01]	0.004	[1, 0]	0.008	0.25 deg	1.05 s	[-10, 10] deg	$-\lvert q^{err}\rvert$

In addition to the shown termination conditions for *options*, controller $h-\gamma$ has an additional termination condition β_T of 25 time-steps. An action produced by this *options* controller steers gamma. Because the time-scale of $\gamma-q$ is equal to 9.41 seconds, the next obtained sample could lie roughly every 1000 time-steps. In an episode lasting 6000 time-steps, 6 samples are gathered. This is a very time-consuming process, and therefore learning is encouraged to happen more often by setting a time-step limit to 25 steps. For all other *options* only β_S and β_B were used.

2. Training Signals and Policy Evaluation

At the start of every episode a new reference signal is generated. A random amplitude is chosen within the parameters' discretisation bounds and sustained for twice the time-scale of the problem. This is repeated until a signal is generated that fills the length of the episode. Step reference training signals were found to be better suited than sinusoidally shaped -frequency-matched- signals. The changing nature of the sinusoid made the agent learning less directed and more stochastic. Figure 7 shows an example of a generated training signal for controller $q-\delta_e$.

Every 12000 experience samples the performance of the agent is evaluated on a set of four test signals. These signals are not experienced by the agent before, and all excite the system differently. They consist of a doublet, 3211-pattern, ramp signal and sine wave. They are defined with unit amplitude and a length of 10 seconds and are scaled in magnitude and time to match the discretisation domains and problem-specific time-scale respectively. The maximum magnitudes of the 3211-pattern and sinusoid are set to half of the discretisation domain magnitude limits. Figure 8 displays the

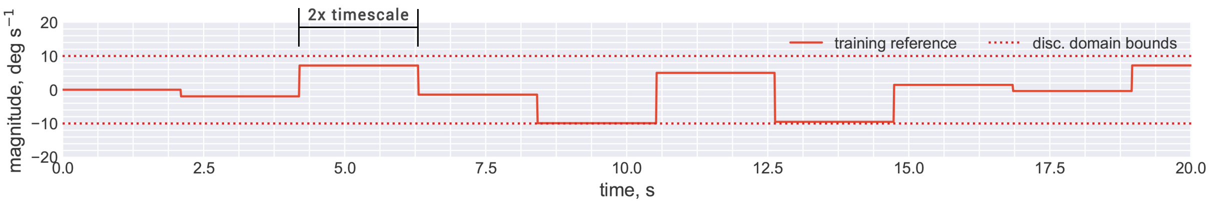


Fig. 7 Example random training signal for controller $q - \delta_e$

evaluation signals for controller $q - \delta_e$, and is of length 10.5 seconds. Evaluation time length for controller $\gamma - q$ and $h - \gamma$ are 94.1 seconds and 177.7 seconds respectively.

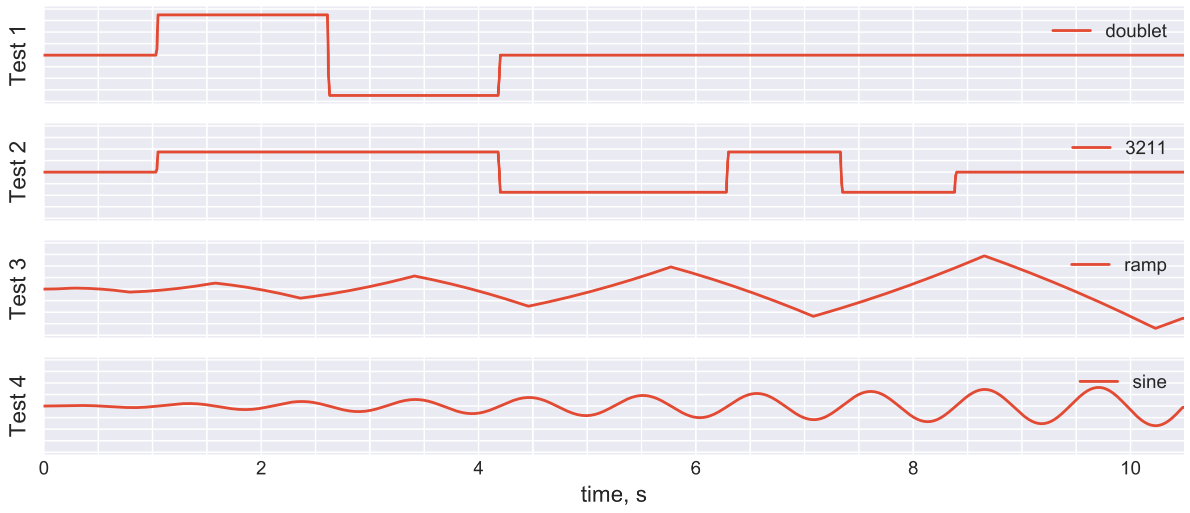


Fig. 8 The evaluation reference test set consisting of four signals; a doublet, 3211-pattern, ramp and sine.

An policy evaluation score is given based on the average tracking error on each of the four tests. For each test the tracking error represents the mean deviation from the reference line as a percentage of the discrete domain. Values closer to zero represent lower tracking errors. A score of -10 within the discrete error space of $-10, 10$ deg would indicate a mean deviation of 10%: 1 deg. Equation 10 shows this for one signal. A visualisation is given in Fig. 9.

$$\text{score} = 100\% \cdot \frac{dt}{T}\left(\sum_{t=0}^{T} \frac{-|\text{ref}_t - \text{obs}_t|}{\max(\text{disc. domain})}\right) \tag{10}$$

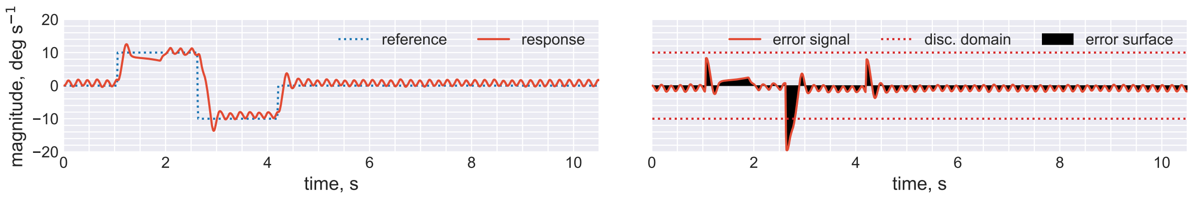


Fig. 9 Evaluation score calculation for Test 1; integration of the tracking error.

Simultaneously with the evaluation of the policy, the change in policy $\Delta\pi$ since the last evaluation point is calculated as the sum of the absolute difference between each entry in the extracted policy table with Eq. 11. Here π is calculated with Eq. 3 and n indicates the n^{th} policy evaluation. By comparing the current change in policy to the maximum change in policy max $\Delta\pi$ encountered during training, a convergence value is computed. If the convergence rises above a threshold value of 98%, the policy has reached convergence and learning is terminated.

$$\Delta\pi_n = \sum_{s\in\mathcal{S}} |\pi_n(a|s) - \pi_{n-1}(a|s)| \tag{11}$$

C. Altitude reference tracking task experiment

To assess the performance of each of the three RL implementations, they are subdued to the same set of altitude reference signals shown in Fig. 10. These are signals neither experienced by the agent in training nor in evaluation. Six altitude reference signals are defined, consisting of ramp signals, step signals and a sinusoidal signal. Period of the sinusoidal reference is one minute. Slope of the ramp signals is set to a realistic value of 4000 ft min^{-1} (1219 m min^{-1}), based on the maximum climb rate of a Cessna Citation 3/5/7 [24]. A performance score is obtained using the evaluation score formula of Eq. 10. The average of the performance is taken as the tracking performance of the controller.

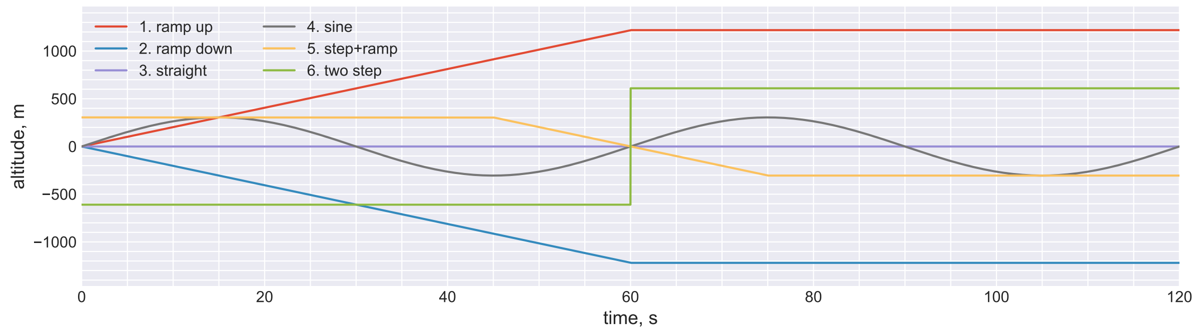


Fig. 10 Six altitude reference tracking tasks for measuring the mean tracking error of each controller strategy.

VI. Results

The results section is split in two parts; learning performance and tracking performance. Learning performance lists the average policy evaluation score and its variance for each problem. Also the learning curves are discussed. To evaluate tracking performance, a comparison is made between each control method on six validation signals. For the hierarchical controller types the learning algorithm is ran five times using the same settings to evaluate the variance in outcome. Of each layer the best performing controller is chosen as the lower hierarchical controller for the next level. FRL is ran only once due to the difficulty of tuning its learning parameters and the lengthy computation time.

A. Learning performance

First the baseline FRL controller is reviewed, strategy **i**. Thereafter the hierarchical controller types HRL and HRL+*options*, strategy **ii** and **iii**. The learning performance is expressed as the evaluation score of the policy over time. Each method is evaluated at a 12000 sample interval. A good learning process should show a clear learning curve and produce roughly the same policies each time. Both hierarchical controllers are compared per controller layer.

1. Flat RL

Because the problem size of FRL is a factor 208 larger compared to the decomposed hierarchical problem structure of HRL, many more samples are required to achieve a policy that is able to complete the evaluation test. Using the 25 discretisations per state and action, a policy could not be found regardless of laborious tuning of the learning parameters ϵ and α. In an effort to still obtain a baseline score, two measures were applied. Firstly the elevator deflection range was reduced, and secondly several lower dimensionalities of FRL were tried of which only the dimension $9 \cdot 7 \cdot 7 \cdot 11$ resulted in a policy. The two measures will be referred to as 'FRL25' and 'FRL97711' respectively, according to their dimension. In short they represent:

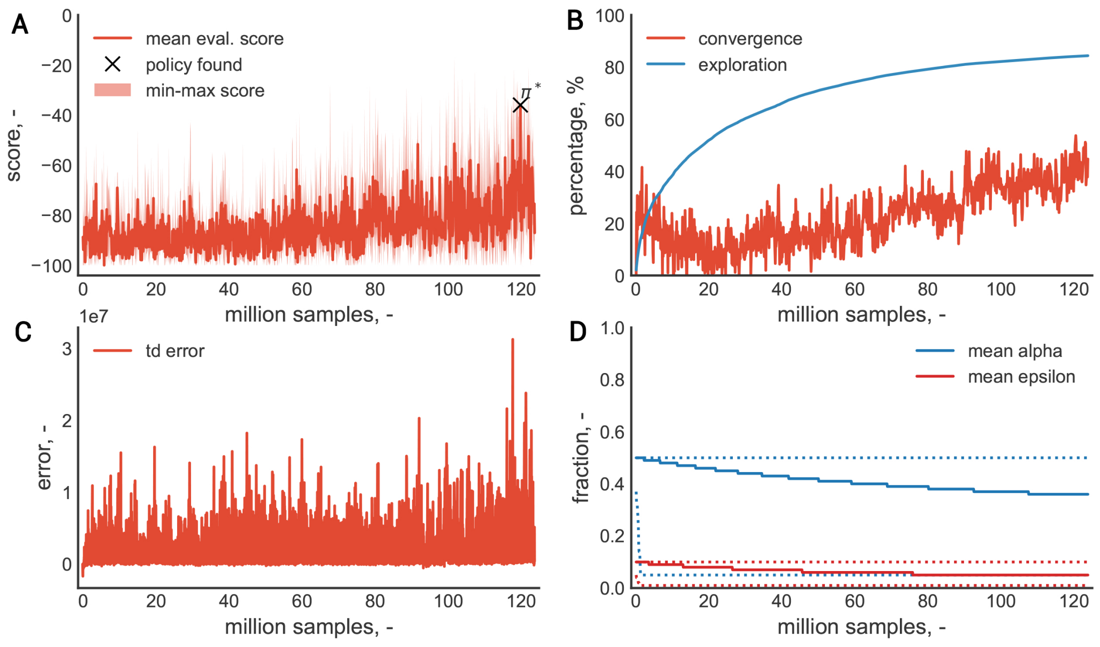


Fig. 11 Learning curve FRL25 with 25 discretisations and δ_e = [-5, 5].

- *Measure 1*, 'FRL25' | A reduction of the elevator deflection range from 12.4 to 5 degrees.
- *Measure 2*, 'FRL97711' | A reduction of the elevator deflection range from 12.4 to 5 degrees, including a reduction of the original 25^4 discretisation space to $9 \cdot 7 \cdot 7 \cdot 11$ (1.24% of the original problem size).

The goal of a smaller deflection angle is to reduce the aggressiveness of direction changes and thereby make it less likely for the agent to accidentally exceed aircraft limits. The goal of a smaller discretisation space is a reduction in problem size, making the problem more feasible to be solved. The maximum achievable tracking performance is decreased by both measures. Firstly *FRL25* is reviewed, thereafter *FRL97711*.

Measure 1, FRL25 | Figure 11 shows the learning process summarised in four graphs. It shows progression of the evaluation score (11A), the level of policy convergence and exploration percentage (11B), the temporal difference error (11C) and the progress of learning parameters ϵ and α (11D). Several observations are made:

- Learning is far from completed. Convergence is at 40%.
- The achieved evaluation score of -37 is not as good as the other methods are able to achieve.
- The policy is obtained at an exploration below 100%, making its action output undetermined in unvisited states.

As there are many combinations of states and actions possible, learning commences very slowly. Initially little about the problem is explored and greedy evaluation results in frequent selection of actions leading to exceeding aircraft limits. As more of the domain is visited the evaluation scores slowly start to increase. After 20 million samples the convergence starts to gradually climb. The td-error however does not show a decrease over time and a clear learning process is not visible here. It is not possible to predict the number of samples that would be required before convergence is reached, but is likely more than double the current amount. Because of the time it takes to reach 100% convergence whilst not having any guarantees the policy will improve further, the current policy and the 119 million number of samples up until the policy are taken to form a baseline score.

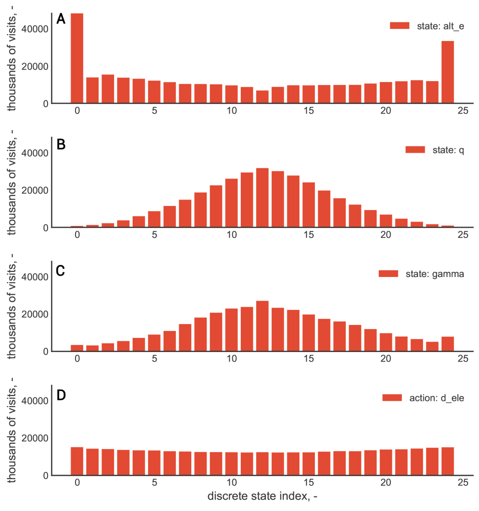


Fig. 12 FRL25 state and action visit information.

Additional information about the learning process of this agent
is shown in Fig. 12, where the distribution of visits for each state and action is shown. From this can be concluded that even though a neutral flight path and pitch rate were visited most often (12B and 12C), the zero-error altitude was visited least (12A). Also no clear choice is made regarding the selection of actions, roughly selecting them equally (12D). This indicates the action selection process does not drive the agent consistently to different areas in the state space. Rather it often selects actions randomly.

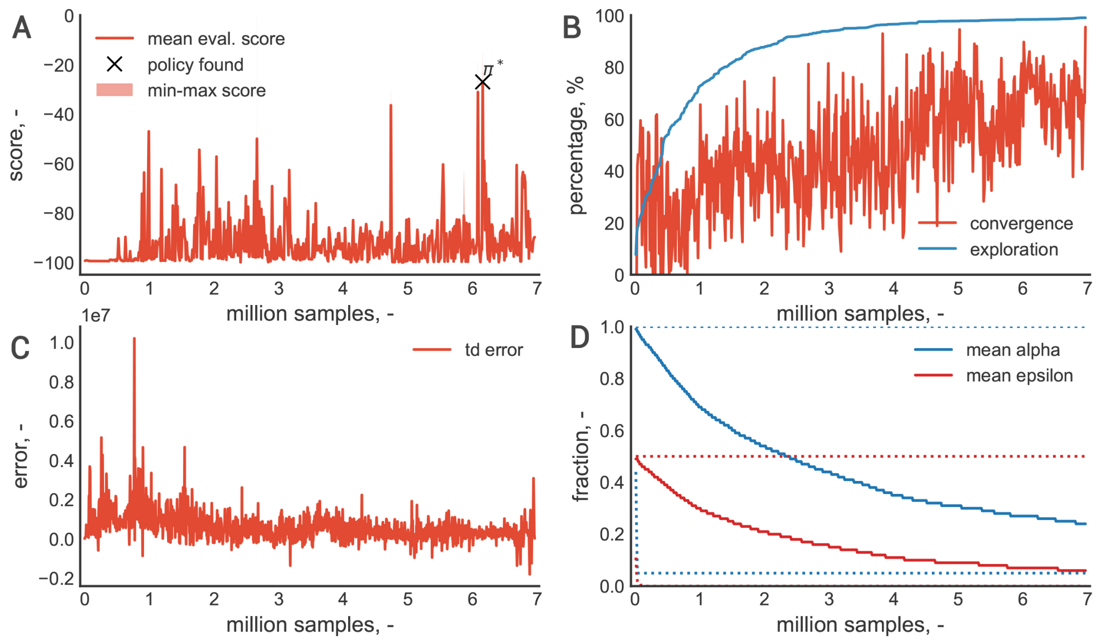


Fig. 13 FRL97711 learning process progression measured at each evaluation interval. Evaluation score (A), convergence and exploration (B), temporal difference error (C) and α, ϵ (D).

Measure 2, FRL97711 | As a second measure the dimension of the problem is brought down in an effort to decrease the number of samples required and to evaluate the effect. Several lower dimensional combinations were tried, but only one resulted in a policy. Its number of discretisations for altitude, pitch rate, flight path angle and pitch rate discretisation number are 9, 7, 7 and 11 respectively. Figure 13 provides the same four-graph summary.

The learning curve with respect to *FRL25* is more stochastic of nature. In the initial evaluations the aircraft exceeds its limits quickly at low exploration of the environment. Thereafter performance sporadically spikes to higher values but only finds a policy near convergence of learning. It achieves a result of -27.0 using 6.96 million samples (13A). The td-error is initially growing and moving further away from zero (13C). This is expected since exploration will cause the agent to select bad actions initially causing it to perform worse. Thereafter this information can be used to improve performance. Exploration signifies the percentage of non-zero entries still present in the Q-table. As it grows the td-error shrinks and the convergence of the policy slowly increases as well. The main issue of this policy is that the learning curve does not show a clear improvement over time, and is chaotic. It could also not be replicated. The resulting policy is therefore not the result of well guided learning and must be regarded as a coincidence.

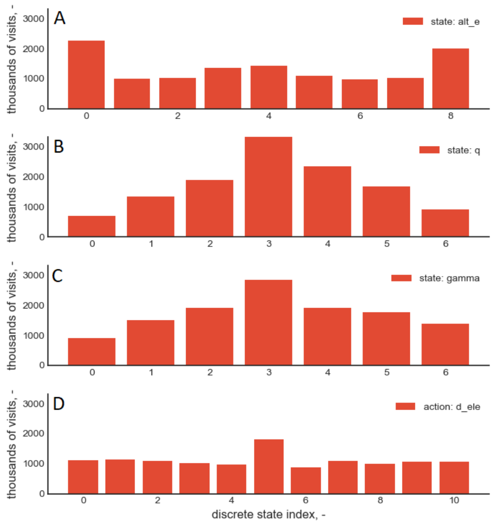


Fig. 14 FRL97711 state and action visit information.

Figure 14 shows the distribution of visited states and selected actions during learning. With respect to the visit ratios of *FRL25*, more learning has occurred near the zero-error altitude (14A). Also the preferred action command is the central one (14D). Curves for γ and q are quite similar (14B and 14C). The reduction in dimension shows that the agent is able to more quickly explore and learn to prefer actions.

FRL25 and FRL97711 | For both measures an important observation is made about the learning parameters α and ϵ, Fig. 11D and 13D. Their mean values decay over time in a smooth manner but their minimum values go to zero almost instantaneously. Managing this trade-off between exploration and exploitation using the visit-dependent ϵ-greedy strategy is difficult. Learning samples are initially spent on states that do not benefit the agent and cause crashing. Because visits accumulate during this process, also the greediness factor increases and learning decreases. This will cause the agent to prefer these states long after the initial phase of learning. Keeping a certain randomness in states often visited during the initial learning phase is important to later have the possibility to escape from this behaviour. A potential solution lies in exploration strategies based on the td-error. Two methods that may be of interest as a future recommendation to mitigate this problem are smart exploration using td-errors [25] and value-difference based exploration [26].

Many attempts of training an agent using either formulation of FRL did not produce a policy at all. They either converged without finding a policy, or got stuck by learning and un-learning behaviour. Repeated learning trials were therefore impossible to perform, further strengthening the argument that FRL is a difficult approach to this high-dimensional problem with large delayed reward. Table 6 shows the evaluation scores and corresponding number of samples for the found policies.

Table 6 FRL: Policy evaluation score and number of samples in millions.

	FRL (25^4)		FRL ($9 \cdot 7 \cdot 7 \cdot 11$)	
Controller	score	samples	score	samples
$h - \delta_e$	-36.0	119.9 M	-27.0	6.96 M

2. HRL and HRL with options

Controller strategies **ii** and **iii** are evaluated per hierarchical layer. Learning each layer is repeated five times with the same learning parameters to obtain a measure of variance between runs. From each five runs the best policy is selected as a controller for the next layer. Table 7 compares the average and standard deviation evaluation scores and samples for each hierarchical layer.

Table 7 HRL and HRL+*options*: Policy evaluation score and number of samples in millions.

	HRL		HRL + *options*	
Controller	score	samples	score	samples
$h - \gamma$	-24.33±1.38	3.67±0.22 M	-28.79±1.96	0.75±0.08 M
$\gamma - q$	-9.82±0.59	4.29±1.37 M	-11.69±0.89	0.43±0.06 M
$q - \delta_e$	-6.93±0.97	2.39±0.53 M	-6.63±0.95	1.27±0.20 M
		10.35±2.12 M		**2.45±0.34 M**

Immediately obvious is the large decrease in the amount of samples when using *options*, most apparent in the second layer. *Options* seems more effective for problems with a larger time-scale, where sustaining an action for a longer period of time is more effective because it compensates for the larger reward delay by persistently driving the agent to a different state. This is in contrast to learning without *options*, where rapidly changing actions result in no effective change of state. Overall however, *options* scores lower across the board and roughly equal on the first layer. This is most likely due to the second layer converging before the best policy is found. Where HRL finds a score of -9.8, *options* only achieves -11.7. The suspicion of early convergence is supported by the number of samples being much lower than the top and lowest layer. Multiple different learning settings where tried to achieve a better result, but many attempts failed. Tuning these parameters is a time-consuming process as explained in Subsection V.B.1, because of the increase in simulation time.

In general then, *options* does not outperform the regular hierarchical control method on the individual evaluation tests. It does however require only 24% of the samples with respect to HRL to form the altitude reference tracking controller. Figures 15 through 17 show the learning performances of HRL over five sequential runs by the mean evaluation score and the minimum-maximum scores. The best policies for each run are indicated.

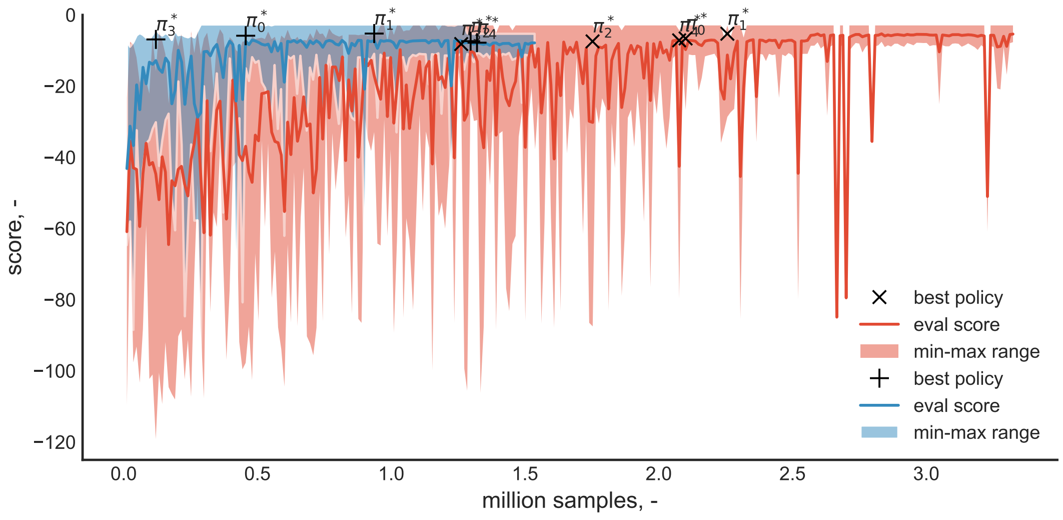


Fig. 15 $q - \delta_e$ **control: Number of samples vs. evaluation score. Average learning performance over 5 runs. HRL in red, HRL plus *options* in blue. Best policies indicated by 'x' and '+' for HRL and *options* respectively.**

From Fig. 15 the speed of convergence of HRL with *options* become apparent. Also the minimum scores produced by *options* are all above those of HRL, meaning the method does not deviate as much during learning but mostly improves in one direction. Not only is the performance better, mostly the initial performance is better. The flat part of the learning curve, where the performance becomes saturated is reached after 33% of its total sample length. HRL reaches this point only after 60% of its sample length.

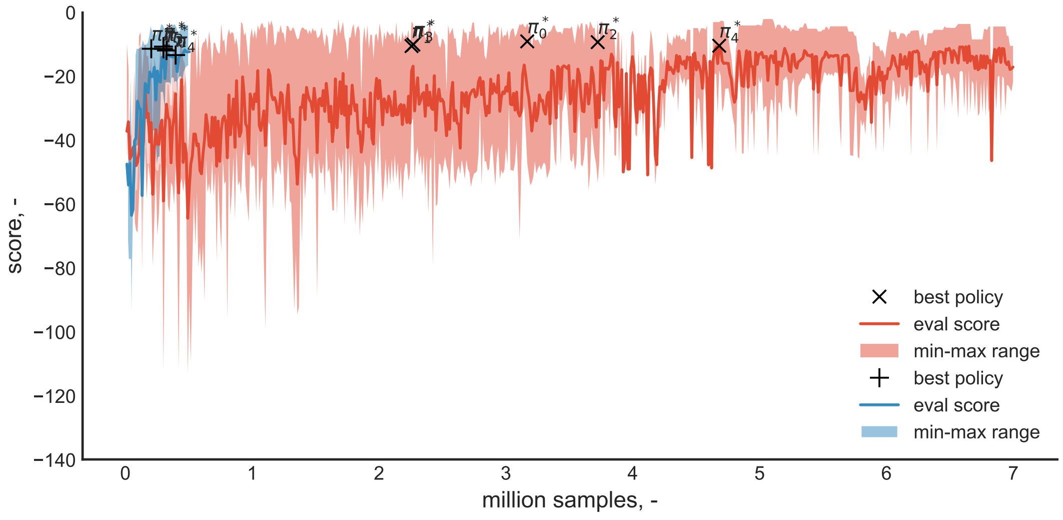


Fig. 16 $\gamma - q$ **control: Number of samples vs. evaluation score. Average learning performance over 5 runs. HRL in red, HRL plus *options* in blue. Best policies indicated by 'x' and '+' for HRL and *options* respectively.**

Fig. 16 shows a different behaviour for the controller $\gamma - q$. Learning shows to be more difficult for HRL to converge by the longer ramp up to the optimal values. Also, the range of the policy evaluation scores is larger than for the first layer. This is not reflected in the outcome of the policies which have the smallest variance of the three layers, as displayed in Tab. 7. Referring to that same table, the variance of the number of samples is large for this layer because one of the five runs was terminated on the pre-set limit of seven million samples rather than converge before this point. The latter part of the graph therefore has a narrower and smaller min-max score range since only one of the five runs is contributing to this part of the graph. Again, the introduction of *options* results in a much faster, less variable learning curve. The difference between HRL and *options* here is largest of the three layers. *Options* does not show a flattened piece in the learning curve, suggesting that indeed this layer might have converged too soon.

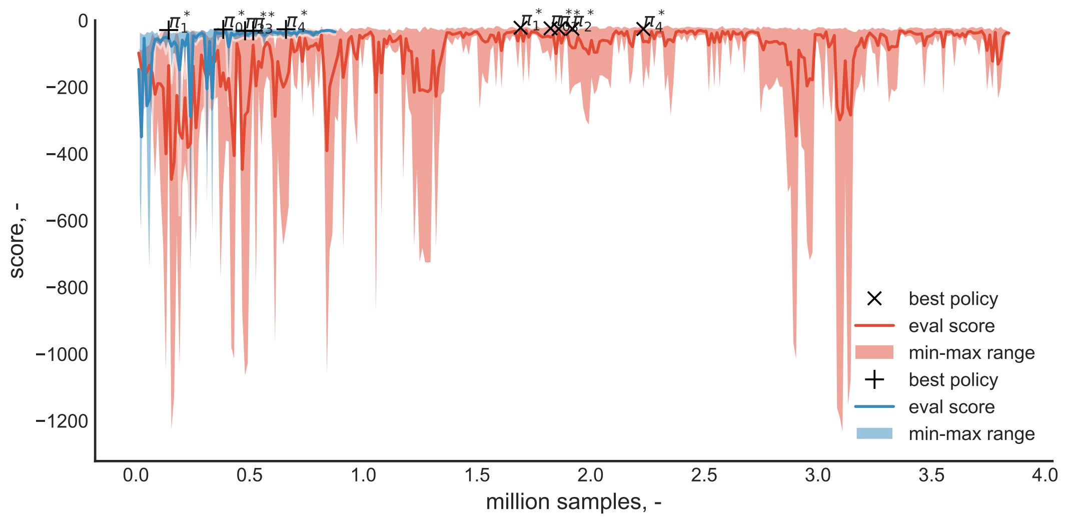


Fig. 17 $h - \gamma$ **control: Number of samples vs. evaluation score. Average learning performance over 5 runs. HRL in red, HRL plus *options* in blue. Best policies indicated by 'x' and '+' for HRL and *options* respectively.**

The learning pattern for controller $h - \gamma$ is different from the other two layers, see Fig. 17. Firstly the score range is much larger because of the used reward signal is in feet, not radians. Instead of steadily converging to a final evaluation score value multiple downward peaks are observed near the end. The policies also do not improve past the midpoint and are concentrated. Rather than converging around 2.5 million samples the agent continues to search. This can either be attributed to the learning parameters not being optimal or the agent having difficulty with learning because of an inconsistent policy from layer two. Also, this is the only control layer where *options* shows clear downward peaks for this control layer. This can be attributed to the same effect of an inconsistent second layer policy, where perhaps certain commands are not satisfied thus limiting the control over altitude as well.

B. Tracking performance

Tracking performance is measured using the policy evaluation scoring of Eq. 10. The average deviation as a percentage of 250 meters is listed for each test in Table 8. The average over six tests and a percentage relative to the baseline is given. Most important differences are underlined. Figure 18 lists the time-traces of each controller per test.

All methods are able to complete the validation test set. They all show the intention to bring the altitude error to zero. Overall performance is best for HRL, achieving an error of only 34% of the baseline. *FRL97711* achieves 46% of the baseline tracking error. The baseline shows largest tracking errors on the first two test signals. From the time-trace of Fig. 18A it can be observed that the agent does make a move in the right direction but is unable to sustain the zero-error state.

HRL and *FRL97711* are very similar in performance except for Test 6. From the time traces shown in Fig. 18B, it becomes apparent that *FRL97711* completes the test but does not satisfy the initial step down. Its performance in comparison to *options* on this test therefore seems better as well since the error area is larger for the slower ascent of *options* than it is to not descent at all. Would the test only have contained a step down *options* would still reach the requested reference, whereas *FRL97711* would not.

Table 8 Tracking error scores on the validation set. Scores indicate the average deviation as a percentage of 250m from the reference line. Percentage relative to baseline performance.

		Test						average	
	Strategy	1	2	3	4	5	6	tracking error	
i	FRL (25^4)	-67.2	-242.6	-26.0	-44.9	-36.6	-206.3	**-103.9**	(100%)
	FRL ($9 \cdot 7 \cdot 7 \cdot 11$)	-15.2	-26.7	-8.9	-41.2	-27.9	-166.1	**-47.7**	(46%)
ii	HRL	-17.3	-30.8	-10.9	-37.9	-25.0	-99.0	**-35.3**	(34%)
iii	HRL+*options*	-17.0	-50.8	-11.5	-67.9	-44.2	-185.6	**-62.8**	(60%)

FRL97711 performs best in Tests 1-3. It shows the effectiveness of a reduced dimensionality on controller performance and feasibility of finding a policy in the first place. However, its reduction is not more effective than the hierarchical decomposition as shown by HRL. Also, it showed to disregard one of the given commands altogether. Furthermore, a comparison of *FRL97711* to HRL and *options* is unfair since they do not attempt to solve the same problem. HRL is a decomposition of the original 25^4 dimensionality, not of a dimensionality $9 \cdot 7 \cdot 7 \cdot 11$. Most importantly however, even though learning *FRL97711* showed convergence, its learning process is stochastic and non-repeatable. Without stable learning it is meaningless to comment on the tracking performance since it is mostly the result of randomness. If the learning process is momentarily regarded as stable, it does show that the discretisation of all variables into 25 parts might not be necessary since a close result can be obtained using less discretisations.

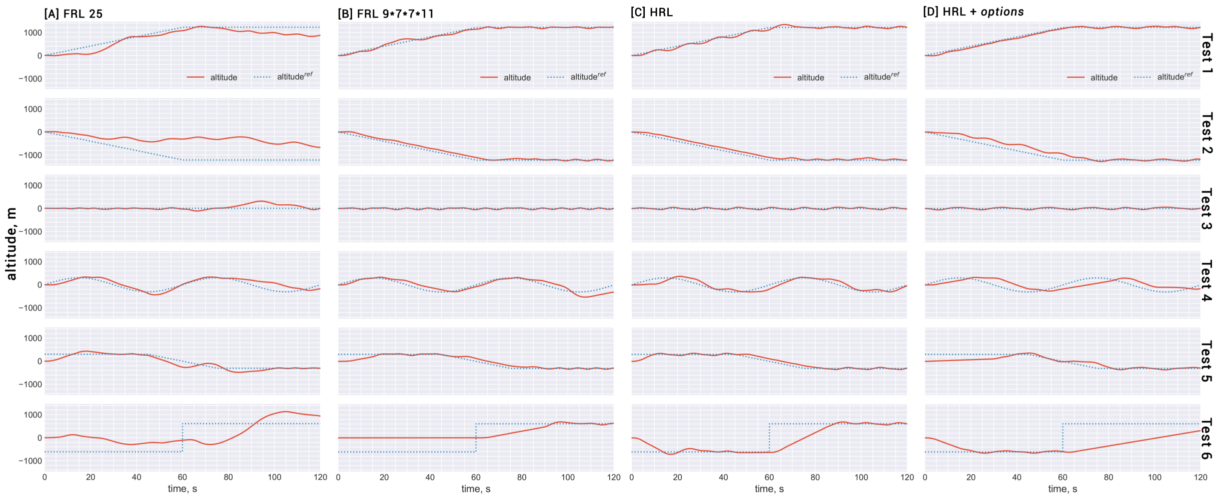


Fig. 18 Validation test performance for [A] FRL (25^4), [B] FRL ($9 \cdot 7 \cdot 7 \cdot 11$), [C] HRL and [D] HRL+*options*.

Because of the large observed difference between methods on the sixth test, more detailed control signals of this test are shown for each controller strategy in Figures 19 to 22. These figures are dissected into four rows and three columns. Each row represents a hierarchical level in the flight controller. From left to right the columns show the given reference command signal and the response, the error signal between reference and the actual state and its discrete value, and finally the generated action output. The generated action from layer 1 appears as reference in layer 2, and the generated action by layer 2 appears as reference input in layer 3. The final row of the figure shows the elevator deflection command and response, the thrust action commanded by the velocity controller, and the velocity of the aircraft. All variables are plotted relative to the trim state of the aircraft.

FRL25 and *FRL97711*, Fig. 19 and 20, have no internal action or reference signals for the flight path angle and pitch rate, since the action output is directly connected to the elevator deflection. This provides little insight into the decision process of the agent. *FRL25* has trouble stabilising the flight path angle and *FRL97711* does not even attempt to start a descent. Overall the internal signals change less over time in *FRL97711*. The fewer discretisations of FRL97711 show a more relaxed behaviour because changes in variables are only detected upon a change of the discrete value.

Figure 21 shows good tracking behaviour by the HRL controller. The given altitude reference command is outside

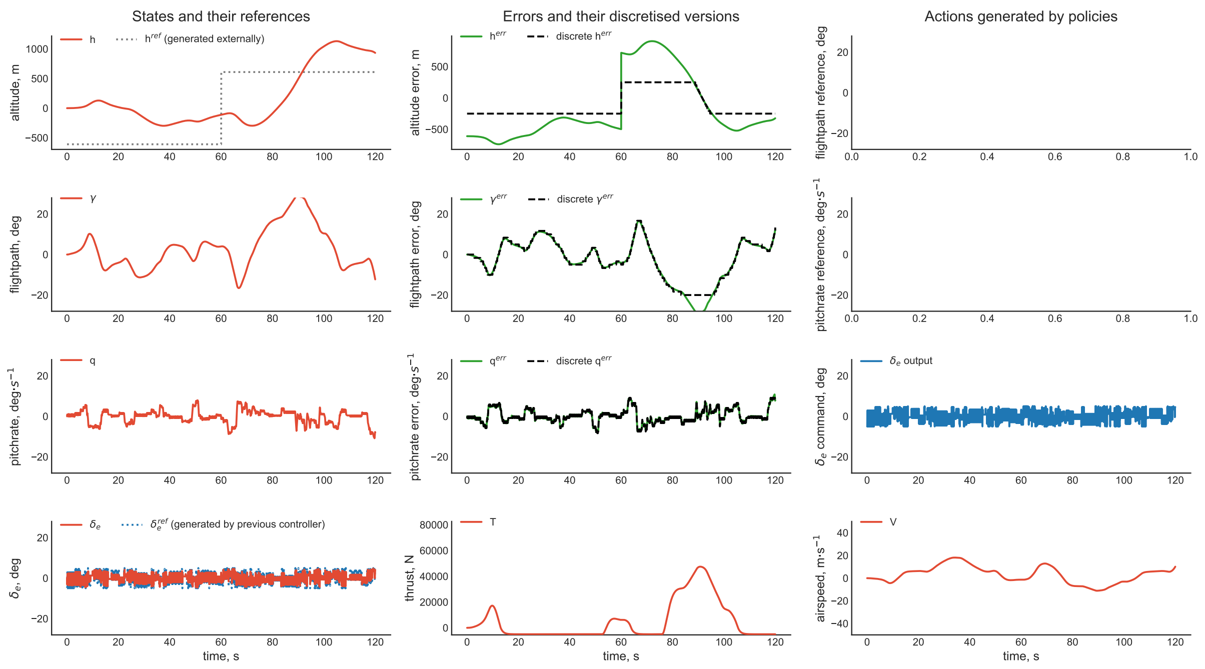


Fig. 19 Tracking performance on validation test six for *FRL25*, showing internal control signals.

of the discretisation bounds of the first controller, but it steers the altitude quickly to within bounds by commanding of the minimum available flight path angle with little overshoot to the zero-error state. Flight path angle commands are satisfied well by the second hierarchical layer, but less quickly changing flight path angle commands would probably benefit the performance further by giving the second layer time to satisfy the given command. All controllers have learned to cooperate and successfully are able to follow an arbitrary altitude reference command.

Less successful in tracking than hypothesised is the *options* method shown in Figure 22. The biggest contributing factor to the lack of performance is controller $h - \gamma$. Although able to produce flight path angle commands of 20 degrees, it chooses not to. Performance of the other layers, $\gamma - q$ and $q - \delta_e$, look less chaotic than the equivalent layers of HRL and have the potential to outperform HRL when given the right commands. It should be investigated how performance of the top layer *options* can be improved.

C. Sample efficiency performance

Table 9 lists the average used number of samples and average tracking error for each of the control strategies. Sample efficiency is compared to the baseline of *FRL25* by comparing the difference in samples and tracking error relative to it. Its sample efficiency is therefore 1. The sample efficiency of the other approaches is calculated as the product of the fractional difference in samples and tracking error.

The largest relative sample efficiency to the baseline method of *FRL25* was found with *options*; 81 times more sample efficient. Its tracking performance is however not better than HRL or *FRL97711*. The sample efficiency of *FRL97711* is higher than that of HRL because even though its performance is worse, it converged in 3 million samples less. As discussed before, the *FRL97711* policy can not be compared to HRL since they solve a different dimensional problem. But mostly its policy is the result of coincidence more than well guided learning. It does however show the effectiveness of a reduced dimensionality and the immediate increase in sample efficiency.

VII. Conclusions and Recommendations

The aim of this research is to explore the application of Hierarchical Reinforcement Learning (HRL) to reduce the number of samples required by flat Reinforcement Learning (FRL) to learn flight control. Thereby making a contribution to the development of model-free adaptive flight control by increasing the sample efficiency of Reinforcement Learning

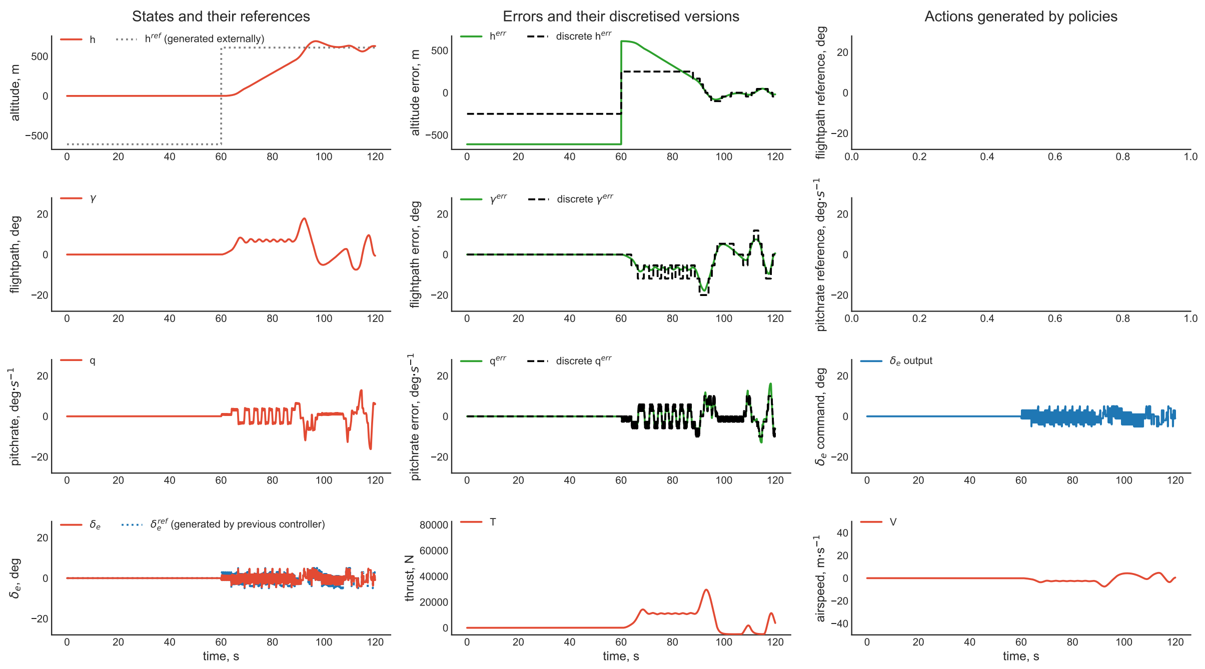


Fig. 20 Tracking performance on validation test six for *FRL97711*, showing internal control signals.

Table 9 Comparison scores of average number of used samples and average tracking error. Sample efficiency is calculated as a fraction relative to FRL, higher is better.

	Strategy		Avg samples		Avg tracking error		Sample efficiency
i	FRL (25^4)		**119900000**	(100%)	**-103.9**	(100%)	**1**
	FRL ($9 \cdot 7 \cdot 7 \cdot 11$)		**6965348**	(6%)	**-47.7**	(46%)	**37.50**
ii	HRL	$h - \gamma$	3674400				
		$\gamma - q$	4289380				
		$q - \delta_e$	2386689				
			10350469	(9%)	**-35.3**	(34%)	**34.10**
iii	HRL + *options*	$h - \gamma$	746495				
		$\gamma - q$	432135				
		$q - \delta_e$	1267582				
			2446212	(2%)	**-62.8**	(60%)	**81.09**

(RL).

The proposed discretisation domain of FRL proves too difficult to learn a policy. To obtain a policy for the problem dimension of 25^4, the allowable elevator deflection is brought down from 12.4 degrees to 5. With this adaptation a baseline policy is found using 119 million samples and a tracking error of -104. Several attempts are performed at showing the effect of dimension reduction within FRL, but only the dimension $9 \cdot 7 \cdot 7 \cdot 11$ resulted in a policy. This reduction results in a sample efficiency increase factor of 37. This performance is however the result of a stochastic learning pattern and a policy that does not show consistent tracking results. It shows the effect of a reduced dimensionality but instability when too little discretisations are used.

While all three control approaches are able to follow the six different reference commands, tracking performance is best for HRL at 34% of the baseline method. Because it requires only 9% of the samples, it achieves a relative sample efficiency factor of 34 to the baseline. The addition of *options* to the decomposed controller has a large impact on the

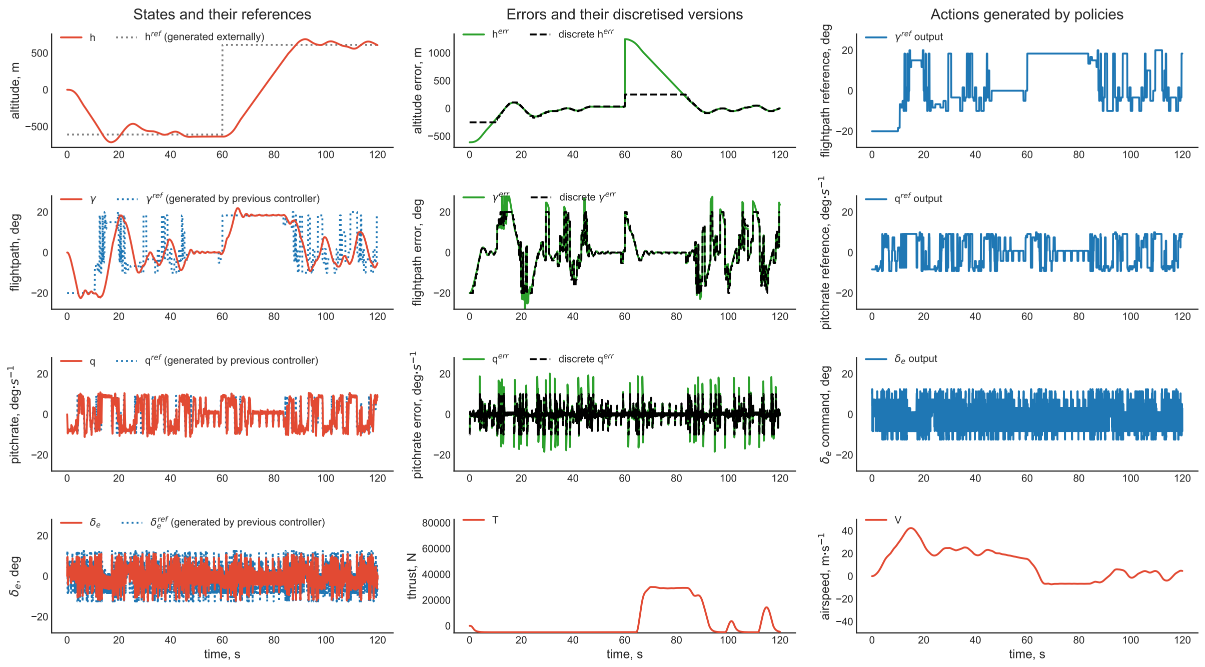


Fig. 21 Tracking performance on validation test six for HRL, showing internal control signals.

sample efficiency and brings the total improvement with respect to FRL to 81 times, making it the most sample efficient. Furthermore, *options* do not make use of eligibility traces. The number of Q-table updates at every update is therefore equal to 1 instead of to the length of the trace. Compared to the maximum trace length of 30 for $\lambda_e = 0.9$ this is a reduction of factor 30. Considering *options* do not update every time-step this reduction increases dependent on the *option* length. Especially for online performance this is interesting since computational resources may be scarce.

The most effective addition over FRL is the hierarchical decomposition in itself. It makes learning of the altitude controller possible with the same four parameters as used in the FRL controller. Where FRL was not able to deal with 25 discretisations per parameter and required a reduced discretisation set to function, HRL was. Additionally, each individual hierarchical control layer can be trained and analysed separately allowing for more optimization. The addition of temporal abstraction of actions shows more confident application of actions with as a result a less noisy controller. It does however not outperform HRL in tracking performance. This is believed due to the second layer ($\gamma - q$) being for both HRL and options hard to learn to control. However, for the same number of samples options is expected to outperform HRL. This would require a different learning setup for which additional experiments are needed.

To improve learning the discretisation domain is scaled linearly towards the centre for two reasons. Firstly this effectively spreads experience samples over a larger part of the discretisation domain and prevents imbalance in learning between states with state-visit dependent learning parameters ϵ and α when concentration of visits in one state can occur. Secondly, it increases the fineness at the zero-error state and thus the potential maximum tracking accuracy that can be achieved. A little discretisation scaling is of help to the learning process, but too much will reduce the efficiency gain again making the centre of the domain harder to reach. A scaling of 0.9 is found to perform best.

Although the design and training of three learning methods is successful, specifically the learning parameters ϵ and α and their progression while learning have a big influence on the outcome. These are costly to tune when training can take several hours to complete. A disadvantage then of splitting the controller in three parts is a tripling of the tunable hyper-parameters. For each of the controller layers the delay in the reward signal and the time-scale of the dynamics is different. This makes it difficult to set global learning parameters that suit each problem when in fact they should be regarded as three separate problems.

This research shows the effectiveness of a hierarchical decomposition and the sample efficiency of options in this framework. More research is needed for further development. This could be directed towards incrementing the operational degrees of freedom beyond longitudinal flight only. Additionally, a static linear dynamical model was used for simulation but it is interesting to discover how the hierarchical structure as a whole can adapt to changes in

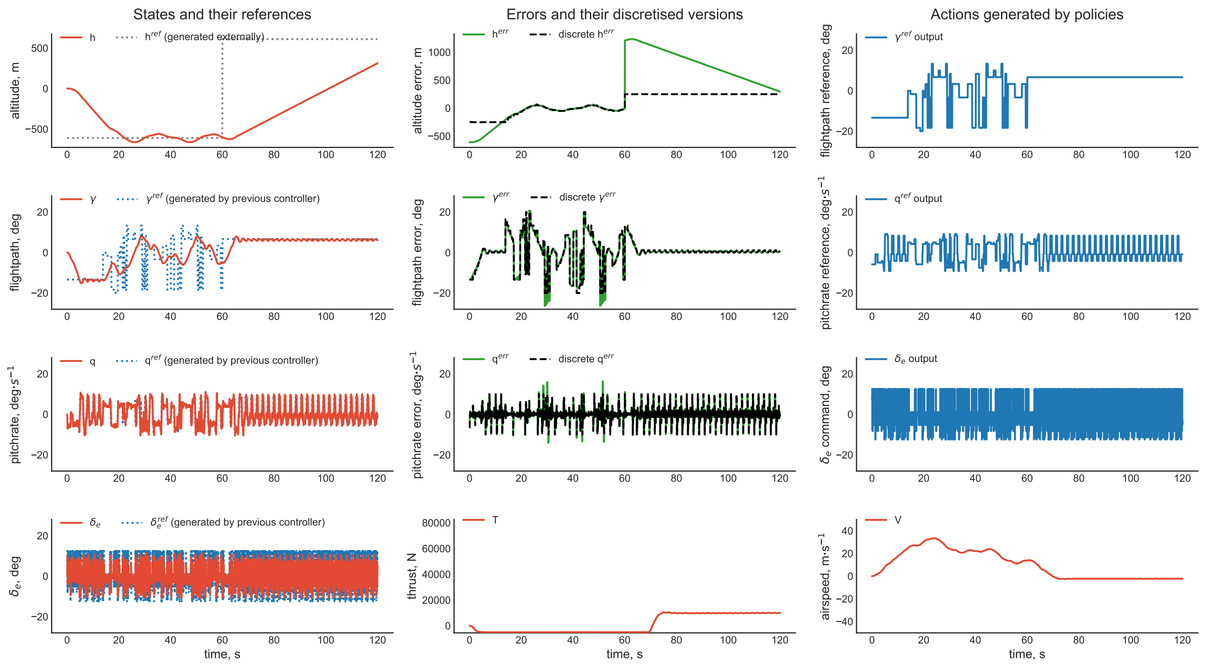


Fig. 22 Tracking performance on validation test six for *options*, showing internal control signals.

the dynamics over time and to non-linearities, which are both possible to be handled with RL. More research can be aimed towards a reduction of the impact of the learning parameters on the outcome. Also, the adaptable character of RL has not been utilised here. An extension to the online setting, where a high sample efficiency is desired to lessen computational load, should be performed.

References

[1] Sutton, R. S., and Barto, A. G., *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge, 1998.

[2] Dietterich, T. G., "Hierarchical reinforcement learning with the MAXQ value function decomposition," *Journal of Artificial Intelligence Research*, Vol. 13, 2000, pp. 227–303.

[3] Bellman, R. E., *Dynamic programming*, Princeton, NJ: Princeton University Press, 1957.

[4] Russel, R. S., *Non-linear F-16 Simulation using Simulink and Matlab*, 2003.

[5] Barto, A. G., and Mahadevan, S., "Recent advances in hierarchical reinforcement learning," *Discrete event dynamic systems*, Vol. 13, No. 1-2, 2003, pp. 41–77.

[6] Sutton, R. S., Precup, D., and Singh, S., "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, Vol. 112, No. 1, 1999, pp. 181–211.

[7] Watkins, C. J. C. H., "Learning from delayed rewards," Ph.D. thesis, University of Cambridge, England, 1989.

[8] Watkins, C. J., and Dayan, P., "Q-learning," *Machine learning*, Vol. 8, No. 3-4, 1992, pp. 279–292.

[9] Sutton, R. S., "Learning to predict by the methods of temporal differences," *Machine learning*, Vol. 3, No. 1, 1988, pp. 9–44.

[10] Singh, S. P., and Sutton, R. S., "Reinforcement learning with replacing eligibility traces," *Machine learning*, Vol. 22, No. 1-3, 1996, pp. 123–158.

[11] Bagnell, J. A., and Schneider, J. G., "Autonomous helicopter control using reinforcement learning policy search methods," *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, Vol. 2, IEEE, 2001, pp. 1615–1620.

[12] Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E., "Autonomous inverted helicopter flight via reinforcement learning," *Experimental Robotics IX*, Springer, 2006, pp. 363–372.

[13] Abbeel, P., Coates, A., Quigley, M., and Ng, A. Y., "An application of reinforcement learning to aerobatic helicopter flight," *Advances in neural information processing systems*, 2007, pp. 1–8.

[14] Coates, A., Abbeel, P., and Ng, A. Y., "Apprenticeship learning for helicopter control," *Communications of the ACM*, Vol. 52, No. 7, 2009, pp. 97–105.

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

[16] Molenkamp, D., Kampen, E.-J. V., de Visser, C. C., and Chu, Q. P., "Intelligent Controller Selection for Aggressive Quadrotor Manoeuvring," *AIAA Information Systems-AIAA Infotech @ Aerospace*, American Institute of Aeronautics and Astronautics, 2017.

[17] de Vries, P. S., and van Kampen, E.-J., "Reinforcement Learning-based Control Allocation for the Innovative Control Effectors Aircraft," *AIAA Scitech 2019 Forum*, American Institute of Aeronautics and Astronautics, 2019.

[18] Mannucci, T., Kampen, E.-J. V., de Visser, C. C., and Chu, Q. P., "Hierarchically Structured Controllers for Safe UAV Reinforcement Learning Applications," *AIAA Information Systems-AIAA Infotech @ Aerospace*, American Institute of Aeronautics and Astronautics, 2017.

[19] Verbist, S., Mannucci, T., and Kampen, E.-J. V., "The Actor-Judge Method: safe state exploration for Hierarchical Reinforcement Learning Controllers," *2018 AIAA Information Systems-AIAA Infotech @ Aerospace*, American Institute of Aeronautics and Astronautics, 2018.

[20] Parr, R., and Russell, S. J., "Reinforcement learning with hierarchies of machines," *Advances in neural information processing systems*, 1998, pp. 1043–1049.

[21] Chentanez, N., Barto, A. G., and Singh, S. P., "Intrinsically motivated reinforcement learning," *Advances in neural information processing systems*, 2005, pp. 1281–1288.

[22] Stevens, B. L., and Lewis, F. L., *Aircraft Control and Simulation*, Vol. 76, John Wiley and Sons, Inc., NY, 1992.

[23] Nguyen, L. T., Ogburn, M. E., Gilbert, W., Kibler, K. S., Brown, P. W., and Deal, P. L., "Simulator study of stall/post stall characteristics of a fighter aircraft with relaxed longitudinal stability," *NASA Langley Research Center, Technical Paper*, Vol. 1538, 1979.

[24] Eurocontrol, "Aircraft performance database," , Oct. 2019. URL `https://contentzone.eurocontrol.int/aircraftperformance/default.aspx?`

[25] Gehring, C., and Precup, D., "Smart exploration in reinforcement learning using absolute temporal difference errors," *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1037–1044.

[26] Tokic, M., and Palm, G., "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," *Annual Conference on Artificial Intelligence*, Springer, 2011, pp. 335–346.

II

Thesis Report

2
Flight Control Design

From its origins of pure mechanical linkage to the current state-of-the-art fly-by-wire systems, the flight control system has made quite the evolution. A mechanical link transfers the forces exerted on the control column to the control surfaces directly. For large transport aircraft or fast military aircraft, the force required to move these surfaces exceed the capability of the pilot [9]. To overcome this, powered control surfaces are used. With respect to the mechanical link, powered control surfaces have the benefit of reducing aerodynamic drag, by eliminating the tabs, and removing control surface flutter which can occur at high dynamic pressures [10]. In doing so, a powered system reduces the role of the pilot to signalling desired control commands, rather than commanding directly. Electrically signalling of control commands allows for the introduction of complex, highly sophisticated functions for enhanced safety and performance [10]. This extension of the flight envelope results in a larger range of dynamic pressure that is encountered by the aircraft, leading to larger changes in aircraft dynamics. To control the aircraft over such an extended flight envelope, the need for a Stability Augmentation System (SAS) arises [11]. The SAS is in place to damp and stabilize the fast rotational modes of an aircraft, since these are the most difficult or impossible to control because of their high frequencies [11]. If the augmentation system is also meant to control the modes in addition to stabilize them, it is called a Control Augmentation System (CAS).

An aircraft is known to have a number of different dynamic modes. These can be divided into symmetrical and asymmetrical types, but can also be split into fast and slow modes. Fast modes are the short-period, roll, and Dutch roll, and involve mainly rotational degrees of freedom. Slow modes are the phugoid and spiral modes, involving mostly translational degrees of freedom [11]. While slow modes can be controlled by the pilot, an autopilot can take over these operations as well as providing more advanced functions such as automatic landing. The system responsible for generating and managing stability and control commands is the Automatic Flight Control System (AFCS). Its tasks are to overcome stability and control deficiency, improve handling or ride qualities, and carry out manoeuvres that the pilot is unable to perform due to accuracy constraints or due to the length of the time over which the manoeuvre extends [12]. Responses generated by an automatic system are much quicker and more accurate than the response of a human pilot. Thus disturbances can be detected and reacted to sooner, preventing unwanted build-up of oscillations. A list of common types of automatic control systems is displayed in table 2.1 [11], divided into function of the SAS, CAS, and the autopilot.

Table 2.1: Aircraft automatic control system types [11]

SAS	CAS	Autopilots
Roll damper	Roll rate	Pitch-attitude hold
Pitch damper	Pitch rate	Altitude hold
Yaw damper	Normal Acceleration	Speed/Mach hold
	Lateral/directional	Automatic landing
		Roll angle hold
		Turn coordination
		Heading hold/VOR hold

Combined these automatic control systems form the AFCS. Among the systems displayed, a certain hierarchy can be denoted, shown in figure 2.1 [12].

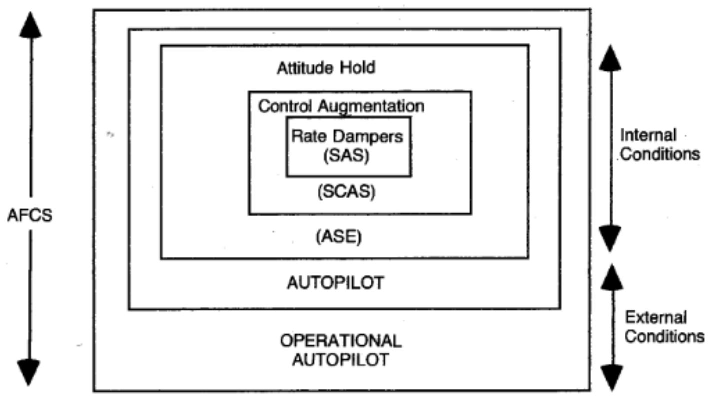


Figure 2.1: Automatic flight control system hierarchy [12]

Here SCAS is the Stability and Control Augmentation System (SCAS), and ASE is the Automatic Stabilization Equipment (ASE) that uses attitude reference input (gyro, compass, etc.) to provide a long-term attitude-hold and heading-hold function [13]. The diagram makes a distinction between internal- and external conditions. Internal conditions correspond to the control loops that are part of the aircraft in the body frame. External conditions involve position relative to the world-frame. Autopilots hold the aircraft to an externally set condition, such as maintaining airspeed and altitude, and operational autopilots perform manoeuvres or series of manoeuvres, such as an automatic Instrument Landing System (ILS) approach, flare and landing [12]. It is the goal of this research to make use of the different forms of hierarchy in aircraft control such as the one present in the AFCS to approach flight control from a machine learning perspective in a more clever way by reusing these elements. The distinctions made between sub-systems in the AFCS is part of a high-level hierarchy in flight control. The remainder of this chapter has the goal of describing the aircraft as a system, as well as discovering hierarchy in lower levels of flight control.

2.1. The Aircraft System

The aircraft, as seen from a control point of view, is a dynamical system onto which forces are exerted that change the state of the system. This general input-output relation that describes the system is a derivation of the real-world physics of the aircraft, a model. As with any model, its physics are described according to a number of conventions about the system. The forces, moments,and control input angles acting upon the aircraft are depicted in figure 2.2. The aircraft of interest is a fixed-wing F-16 fighter jet, of which a detailed non-linear simulation model is available for this research [8].

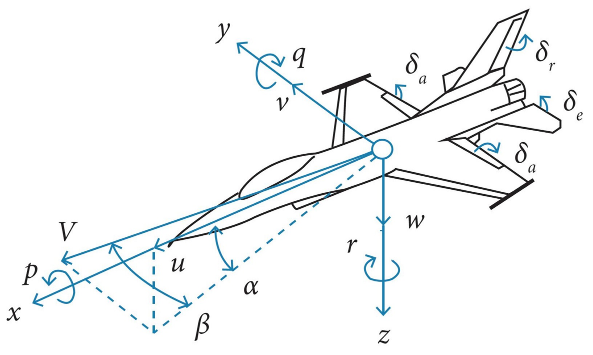


Figure 2.2: Aircraft forces, moments, and control input angles conventions [14]

Four control inputs, eighteen output states, a leading edge flap deflection and a model flag make up the aircraft model. The flag determines use of either the high-fidelity model [15], or the low-fidelity one [16].

The leading edge flap (not in the diagram) is a device to enable the F-16 to fly at higher angles of attack. Its deflection is automatically set in relation to the angle of attack and only available in the high-fidelity model. More information about the exact differences between low- and high-fidelity is found in the included document of [8]. Four inputs to the system are three control surface deflections and a throttle set-point. The eighteen output states of the system are: north and east position, altitude, three body rotation angles, angle of attack, side-slip angle, three body rotation rates, three normalized accelerations, Mach number, dynamic pressure, and static pressure. The system inputs and outputs are formulated here according to Reinforcement Learning (RL) convention as actions (a_t) and states (s_t) in time respectively:

$$a_t = [\tau, \delta_e, \delta_a, \delta_r] \tag{2.1}$$

$$s_t = \left[n_{pos}, e_{pos}, h, \phi, \theta, \psi, V_t, \alpha, \beta, p, q, r, a_{nx}, a_{ny}, a_{nz}, M, \bar{q}, P_s\right] \tag{2.2}$$

The system outputs changes of the state vector (state derivatives) in response to the actions applied to it, as would be the case for a typical state-space system. By integrating the derivatives, the values of states over time can be tracked. The system is given as a diagram in figure 2.3, where the input size of four (4) corresponds to the number of actions, and the output size of eighteen (18) to the number of states.

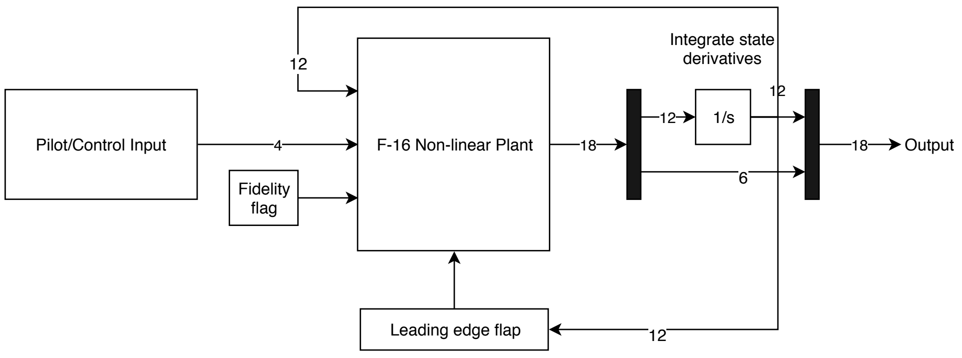


Figure 2.3: Aircraft system I/O diagram, adapted from [8]

The last six states of equation 2.2 are an instantaneous quantity, and therefore do not require integration. Furthermore, the leading edge flap sub-system regulates deployment of the leading edge flap based on the states of the aircraft. Finally the fidelity flag (constant) indicates use of the fidelity of the model. Control of this aircraft takes place in the 'Control Input' section, where control commands are generated based on the states of the aircraft, the current goals, and the implemented control strategy. This is where the AFCS is housed, and the part of the system this research is focussed on.

2.2. Feedback Control

Automatic flight control modes use a type of control algorithm to perform a desired goal. For stabilisation of the aircraft, the general idea is to use feedback control. Here the output of the system is used to control the input to the system, thereby forming a closed loop control structure. This is called state feedback. By using negative feedback the output of a plant can be regulated to hold a certain set-point, turning the task into minimization of an error signal. By changing the reference point, the control loop can also be made to track a signal. Both tracking and regulating are essential functions of control. A single feedback control loop is shown in figure 2.4 [11].

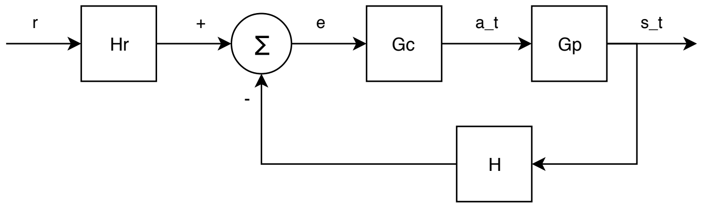


Figure 2.4: Single loop feedback control diagram, adapted from [11]

Here G_c and G_p are the controller and plant respectively. The block H_r is the control pre-filter, that ensures the right physical quantities are compared in the summation. The H block may represent the transfer function of sensor behaviour when measuring the system state s_t. By changing the gains present in the controller G_c, the system can be tuned to behave in a certain way. Feedback control is used throughout in the automatic aircraft controller. As an example consider a roll-angle-hold subsystem of figure 2.5 [11]:

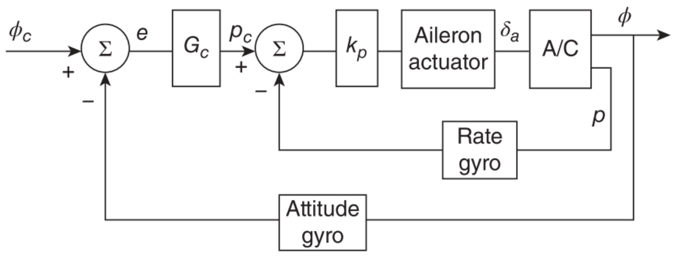


Figure 2.5: Roll angle hold control system [11]

The goal of this subsystem is to maintain a desired roll angle, as set by the reference. By sensing of the current attitude, the first step of the system is to convert this signal into a desired roll rate. After summation with the current roll rate, as sensed by another sensor, the inner loop control converts the roll rate error into an aileron deflection. A system like this makes use of a *cascaded* controller structure, where the output of one controller is the set-point of the next. Structures of this kind are in place to add logic to the controller. A change in roll angle is logically achieved through a change of the roll rate, which in turn is achieved mostly by changing the angle of the aileron. By expressing these physical relations in the controller, each component of the controller is more meaningful and can be tuned to alter a specific part of the larger functionality.

2.3. Hierarchy in flight control

Subsystems like the roll-angle-hold system in figure 2.5 are essential to flight control. Depending on the mission and states of the system, one or more subsystems may be active at the same time. Examples of other subsystems are the functionalities of the AFCS from table 2.1, but can also be found in the book by Stevens and Lewis [11]. Selection of the active subsystems at a given time has to come from a higher level. A *state machine* is a high-level controller that defines different states the system can be in, and describes transition constraints for moving to another state. Some examples of high-level states are; cruise, loiter, and land.

Each of the subsystems have to be tuned to perform optimally in a flight condition. This is achieved through open-loop analysis to find appropriate gains for closing the loop. It is very difficult to design a controller for a non-linear system directly. Therefore a common approach is to apply *gain scheduling*. The system is locally linearised on a multitude of operating points. Around each operating point the system will approximately act linear. A table of appropriate gains is calculated and stored for the controller on each operating point. Controller gains are then adapted in-flight according to the current flight condition. Via this procedure, flight control is extended over the entire non-linear flight envelope using linear control techniques.

Different forms of hierarchy in flight control have been discussed. State machines select the main subsystems that are active at a given time based on the mission and states of the aircraft. One level down subsystems are in place to divide control of the aircraft into smaller, more manageable pieces of control. Scheduling of the gains in subsystems is another level of the hierarchy that controls the correct functioning of the subsystem in that regime. At the lowest level of flight control is the sub-division of the subsystems into inner- and outer control loops. There, each of the control loops have their own reference tracking task, and further divide the problem into smaller sub-problems. These hierarchical shapes of the flight controller give structure to the problem of flight control design, and is an important piece of controller design. The other step being finding appropriate gains for the controllers, and provide switching rules for each of the branches in the hierarchy. The hierarchical ideas of conventional flight control as discussed in this chapter are of importance in chapter 4 when discussing hierarchical self-learning control.

2.4. Controller design

In the previous sections controller gains were mentioned as a large part of controller design, next to the structure of the controller itself. For finding the appropriate gains for a controller different methods exist.

2.4.1. Conventional control

Conventional control methods are commonly regarded as control methods appropriate to Linear Time-Invariant (LTI) Single-Input Single-Output (SISO) systems [9]. The essence of this classical design is a one-loop-at-a-time design approach guided by experience for selection of the control structure [11]. These classical methods tune a system in single steps starting from the inner loop, working inside-out, towards the outer loop. Here design methods such as the Bode and Nyquist plot, and the root locus are used. Modern control design is aimed at simultaneous calculation of suitable controller gains, rather than successive, closing all loops at the same time [11]. The approach of modern control is applicable to Multi-Input Multi-Output (MIMO) systems, as is the case for aircraft systems, instead of just SISO ones. A performance criterion is specified with which a solution can be found for the gain matrix K, solving for all gains at once.

Since the aircraft is inherently a non-linear type of system, the control design methods either make use of (local) linearisation of the system after which linear control laws can be applied, or they use non-linear control laws. A hybrid solution to this problem is the use of dynamic inversion, or feedback linearisation. Dynamic inversion is a technique that uses on-board dynamic models and full-state feedback to globally linearise dynamics of selected controlled variables [17]. This procedure circumvents the need for gain scheduling, since the system acts globally linear. Simple linear controllers can then be designed to control the system. Variants of the dynamic inversion method are increasingly more popular. Incremental non-linear dynamic inversion has shown to be a robust control method and a relatively easy method to be implemented on new types of aircraft using a simple aerodynamic model [18].

2.4.2. Adaptive control

A commonality among the mentioned design approaches is the need for a system model. This assumes the availability of an accurate model of the dynamics of the aircraft. In practise it is very difficult to obtain a highly accurate model [11]. The model is therefore not a perfect representation of the actual behaviour of the system. This limits the performance of a controller to the quality of the model. A possible solution to this issue is the use of adaptive control [19]. Adaptive controllers can apply automatic adjustment of gains on-line. An extension of this idea is to completely remove the dependence on a model in the design of a controller. Methods of this kind are called *direct* or *model-free*. A direct model wouldn't require a system model, and thereby circumvents the problems associated with it. This is especially useful in applications where modelling is difficult because of the complex non-linear dynamics, such as in flight control. Other advantages of a model-free technique is that it is easier to create a control system for a new type of aircraft. In case some dynamics change in the process of the development of the aircraft, the model-free approach does not require the construction of an updated model.

In addition to the need for an adaptive controller, the recent trend towards autonomous vehicles motivates a need for more intelligent control as well. As a control system is given increasingly more control over the entirety of the system, the need for autonomous decisions based on observed system states arises. This extends beyond just the adaptiveness of a controller within bounds of its operating range. A controller needs the ability to learn from experience to overcome the issue of encountering unforeseen circumstances. A promising method that allows for all of these requirements moving forwards in the area of future aircraft control is Reinforcement Learning (RL), a direct adaptive optimal control method [20]. Some of the advantages of an intelligent system are; enhanced mission capability, improved performance by learning from experience, increased reliability and safety of flight [21]. In this context reinforcement learning is promising, but does suffer from sample inefficiencies for large problems such as flight control. Because of the mentioned advantages and potential of RL in flight control, this thesis is aimed at discovering how current flight control methods can be used in the realm of RL to produce a more efficient self-learning controller.

2.5. Flight conditions

It is important to define the boundaries in which the aircraft of this research has to operate. Firstly this makes comparison of different control algorithm on the same platform possible, and secondly this limits the scope of the research to within clearly set boundaries. Since the aircraft considered is a military type aircraft it would be logical to consider military flight operation conditions in the simulation. Flight manoeuvres of this type are however highly-non linear and would be difficult to design a controller for, and a different topic of research in flight control. The focus of this research will rather be on how reinforcement learning can make use of the controller structures present in flight control. Since it is more important to make a good comparison between implemented control methods on the same aircraft than it is to provide sophisticated highly non-linear control, civil aviation operation conditions will be considered only. Since the simulated platform can operate over a large dynamic range, it is possible to let such an aircraft fly manoeuvres that require less performance then it is designed for. Vice versa would be impossible: a civil airliner cannot perform like a military fighter jet. To only fly civil manoeuvres, instead of military ones, will therefore not diminish the results of the research, but rather not unnecessarily complicate matters. The set of typical flight conditions that are encountered in civil flight are [10]:

- taxiing
- take-off including take-off run and rotation
- climb under different conditions where thrust and airspeed vary
- cruise with minimum direct operating costs
- turns (typical bank angles less than 30 deg)
- descent with idle or reduced thrust
- approach (non-precision approach, 3 deg ILS approach etc.)
- go-around in case of a missed approach
- flare
- roll out

On top of these general flight conditions, the aircraft flight envelope is limited by certification to safety limits. While these limits define a certain flight envelope, typically flight only takes place in a smaller part of the allowed envelope. Some important limits of the nominal operating conditions are [10]:

Table 2.2: Nominal flight operating condition limits

Limit factor	min	max	unit	condition
Load factor n_z	-1.0	2.5	g	clean
Load factor n_z	0.0	2.0	g	slats and/or flaps out
Load factor n_z	0.85	1.15	g	passenger comfort
Bank angle	-30	30	deg	
Airspeed V_t	$1.2V_S$	V_{MO} or M_{MO}	m/s	landing
Airspeed V_t	$1.3V_S$	V_{MO} or M_{MO}	m/s	take-off

Where V_S is the stall speed, and V_{MO} and M_{MO} the maximum operating speed and Mach number.

3 Reinforcement Learning

Reinforcement Learning (RL) is one of three classes in Machine Learning (ML), together with unsupervised- and supervised learning. All three are aimed at different aspects of learning from data. Unsupervised learning is useful for clustering data, i.e. for finding patterns in datasets. Supervised learning is useful for classification, labelling of data into the correct category, and regression, estimating relations of variables. Reinforcement learning combines techniques from both of these types of ML, but is concerned with learning from interaction. In interactive problems examples of desired behaviour can not be expected to be available (as is the case with supervised learning) [1], so an agent must be able to learn from its own experience. Also, an agent is learning to maximize a numerical reward signal rather than trying to find a hidden structure (as in unsupervised learning). RL is therefore useful for application in control problems, where through interaction with a system a suitable control strategy can be learned. The focus of RL is to find an optimal strategy for a task without explicitly being told how to do so.

This chapter aims to give an overview of the theory behind reinforcement learning, how it emerged, and in what context it is being applied. Results that have been achieved with RL in the past, and the current state-of-the-art are touched upon. This theory is used in preparation for chapter 4, where the hierarchical extension of RL is discussed.

3.1. Context and applications

Reinforcement learning has emerged as a combination of trial-and-error learning, optimal control, and temporal-difference methods [1]. The earliest and one of most popular techniques in RL, Q-learning, dates from the year 1989 [22]. Afterwards, developments in the field started to gradually reach more mature forms. Tesauro's TD-Backgammon algorithm from 1995 shows how RL can be used to perform human-like moves in the game of Backgammon, performing extremely close to equalling the worlds best players [23]. Crites & Barto used RL to control the dispatching of an elevator in 1996, outperforming the best algorithms of that time [24].

Reinforcement learning proved however difficult to scale to more complex problems. To learn more efficiently, researchers developed adaptations and enhancements to the basic RL framework. An important goal was to introduce generalisation and abstraction in the learning process to stray away from tabular solutions that are impractical and inefficient in many large problems. One such direction in these developments is Hierarchical Reinforcement Learning (HRL), introducing a layer of hierarchy in the learning process. From 1998 until 2000 the three main forms of HRL where formalised. Parr & Russell described Hierarchies of Abstract Machines (HAM) [25], Sutton formalised options [4], and Dietterich developed MAXQ [2].

In the years thereafter things were more quiet. Big breakthroughs haven't been made until after the rise of deep learning in machine learning in 2013. Deep learning is a big step in enabling RL to scale to problems previously intractable, meaning high-dimensional state-action spaces. The properties of deep neural networks to perform function approximation or learn a compact representation of a high-dimensional problem make this possible. Famous examples of applications that made successful use of deep reinforcement learning are the Atari games (2013) [26], the game of GO (2016) [27], and chess (2017) [28]. Nowadays deep reinforcement learning is the state-of-the-art in research involving large complex problems.

Simultaneously, the available tools for the development of RL algorithms has grown in recent years. According to the researchers at OpenAI, RL is slowed down by the lack of available benchmarks and lack of standardization of environments used in publications. Therefore the OpenAI gym library was made public in 2016 to provide researchers with common problems. In combination with other code libraries such as Tensorflow and Keras it is much easier not to start from scratch each time. The combined efforts make reinforcement learning more accessible and an interesting field of research for complex problems such as flight control.

3.2. Elements of Reinforcement Learning

The standard reinforcement learning framework is a process modelled as a finite discrete-time Markov Decision Process (MDP) [3]. Named Markov, since it satisfies the Markov property, where the future is independent of the past given the present. An important property because it allows to summarize all events leading up to the current state simply as the current state. An MDP consists of [29]:

- a set of states $\mathtt{S}$
- a set of actions $\mathtt{A}$
- a reward function $\mathtt{R}:\mathtt{S}\times\mathtt{A}$
- a state transition function $\mathtt{T}:\mathtt{S}\times\mathtt{A}$

When advancing through discrete time ($t = 0,1,2,3,...$) a transition from state s_t to the next state s_{t+1} is made using action a_t. Which state the process transitions to is dependent on the state transition function $T(s,a,s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$, the probability of transitioning to state s' if action a is chosen from s. Note that the state transition function $T(s,a,s')$ does not depend on the time at which the decision is made. A two-state example MDP is given in figure 3.1, where from each state two actions are possible. The transition probabilities after taking an action are given on the dotted lines:

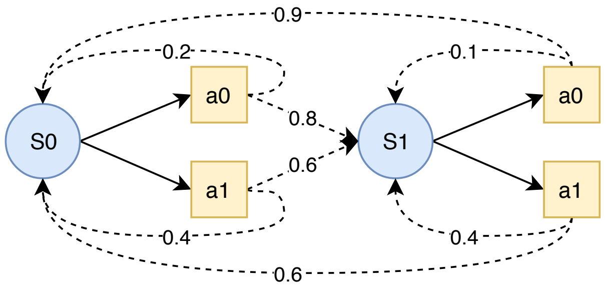


Figure 3.1: A Markov decision process visualized. Probabilities of state transitions after choice of action a_0 or a_1 given as fractions.

In this example choosing action a_1 from state s_1 has a 40% change of resulting in state s_1 again, and a 60% chance of transitioning to state s_0. With each step in the process a state transition is caused by performing an action. Additionally, each state transition is coupled to a numerical reward that is received by the decision maker as a result of the action it has chosen to perform. This reward r_{t+1} is received immediately after a state transition has occurred. For *episodic tasks*, where the reinforcement learning problem naturally breaks down into subsequences, the total reward received in one of these episodes is simply the sum of rewards from each time-step:

$$G_t \doteq r_{t+1} + r_{t+2} + \cdots + r_T \tag{3.1}$$

Where T is the time of termination of the episodic task. Examples of episodic tasks are a level in a video game, chess, and maze navigation. An episodic task ends in a state called the *terminal state*. Learning is done by running episodes over and over, resetting to a starting state at the start of each episode. The start of the next episode is independent from the finish of the previous one [1]. For *continuing tasks*, where the final step time T goes to infinity, the return formulation of 3.1 can potentially grow to infinity as well. To account for this effect the sum of rewards is *discounted*. Each next reward is multiplied by γ to lessen the influence of future rewards. This discount rate γ is a number between 0 and 1 ($0 \le \gamma \le 1$) where values of $\gamma = 0$ would only consider immediate rewards (myopic), and $\gamma \to 1$ would more strongly include future rewards (far-sighted):

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.2}$$

It is the objective of a decision maker to select actions so as to maximize the expected discounted return. By advancing through time, continuously selecting actions and observing the reward signal, the decision maker increasingly becomes more knowledgeable about the process. This experience can be summarized as the result of being exposed to a repeating chain of states, actions and rewards through time:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{3.3}$$

This sequence inspired the name for SARSA, next to Q-learning a popular algorithm in RL. Becoming more knowledgeable means that better actions are selected in each situation based on experience gained previously. These actions are selected by the decision maker according to its policy $\pi(a|s)$, a mapping from states to probabilities of actions. To meet the objective, the decision maker has to learn a policy for which the sum of expected discounted reward is maximized. It is then said to have obtained the optimal policy $\pi^*(a|s)$ for the problem. In reinforcement learning the decision maker is called an *agent*, and the process that produces rewards and state transitions is called the *environment*. RL methods specify how an agent's policy is changed in response to the experience it has gained [1]. The agent-environment interaction is shown in figure 3.2 [1]:

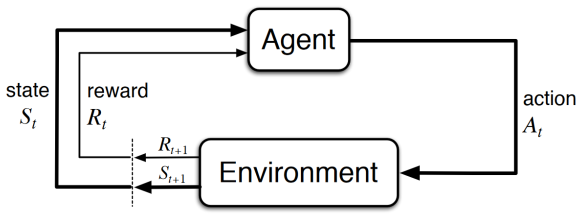


Figure 3.2: Elements of Reinforcement Learning, displaying the interaction between the agent and the environment [1]

3.3. The Bellman Equations

With the elements of reinforcement learning known, the next question becomes how to solve the MDP. This question is equal to finding the optimal policy $\pi^*(a|s)$ that maximizes the expected discounted reward. To get a value for the expectation of this discounted reward, two types of *value functions* are used; the state-value function ($V(s)$) and the action-value function ($Q(s,a)$). The state-value function returns a value of how good a certain state is to be in. It expresses the expected sum of discounted rewards the agent will receive under policy π starting from this state. By expanding the expectation operator of equation 3.4, a summation over probabilities of states, actions, and rewards emerges. Equation 3.5 is known as the Bellman equation for the state-value function:

$$\begin{aligned} V^\pi(s) &= E_\pi[G_t \mid s_t = s] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \qquad (3.4) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V^\pi(s')\right] \qquad (3.5) \end{aligned}$$

Here $p(s',r|s,a)$ represents the state transition function $T(s,a,s')$, the probability of ending up in state s' and receiving reward r given current state and action s,a. Similar to the above expansion the Bellman equation for the action-value function can also be obtained. This equation gives a measure about how good a certain action is to take from a state, when thereafter following policy π. Equation 3.7 is the Bellman equation for the action-value function:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi[G_t \mid s_t = s, s_t = a] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, s_t = a\right] \qquad (3.6) \\ &= \sum_{s'} p(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')\right] \qquad (3.7) \end{aligned}$$

Both Bellman equations (3.5, 3.7) [3] express the current state- and action-value in terms of the next, a recursive formulation. It is known that optimal policies share the same state-value function $V^*(s)$ and the same action-value function $Q^*(s,a)$ [1]. The goal therefore becomes to find the optimal value functions using the Bellman equations. These optimal value functions are known as the Bellman optimality equations:

$$V^*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V^*(s')\right] \tag{3.8}$$

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} Q^*(s',a')\right] \tag{3.9}$$

If either of the optimal value functions is known, selecting actions greedily with respect to their values results in an optimal policy. For the state-value function this implies performing a one-step lookahead to the next highest state-value and selecting that action. This would require knowledge of the state transition function to do so. With the action-value function it is simply a case of finding its maximum, since the value of actions are already contained in the function.

3.4. Dynamic Programming

The search for an optimal policy using the Bellman equations is a problem that can be solved with Dynamic Programming (DP) [30]. DP is a collection of algorithms able to solve problems in which the MDP is known. DP is based on the principle of optimality [3], which implies that an optimal policy can be constructed in a piecewise manner. If the solutions are optimal for each subproblem, then so is the main problem. To make DP work, the problem must satisfy two properties: those of *optimal substructure* and *overlapping subproblems*. Markov decision processes possess both; the Bellman equations recursively decompose the problem, and the value function stores and re-uses found solutions. Most approaches to solve a reinforcement learning problem are closely related to DP [31]. This section will explain the routines of DP before extending to RL.

In dynamic programming full knowledge of the MDP is assumed. This knowledge is used for planning in the MDP. Two types of problems are solved with this: a *prediction* problem and a *control* problem. Prediction is the problem of evaluating the value of a policy, control is the problem of finding the optimal policy. *Policy evaluation* and *policy improvement* are two distinct procedures in the search for the optimal policy. Evaluation of the policy is performed by determining its corresponding state-value function. Improvement of the policy is achieved by making *greedy* action selections with respect to the value function. The action leading to the highest expected return according to the value function is selected always. For each value function then, a corresponding policy exists. A value function that has a higher expected return results in a better policy. And from a better policy better actions can be selected, leading to higher return values. By repeating this process eventually the optimal policy is found. This repeating chain of updates eventually leads to the optimal policy [1]:

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \tag{3.10}$$

Finding the optimal policy is thus a case of finding the optimal value function. The interleaving of policy improvement and policy evaluation is known as Generalised Policy Iteration (GPI). Graphically put, figure 3.3 [1] shows that eventually one will arrive at the optimal solution since both quantities improve closer to their optimal values over time.

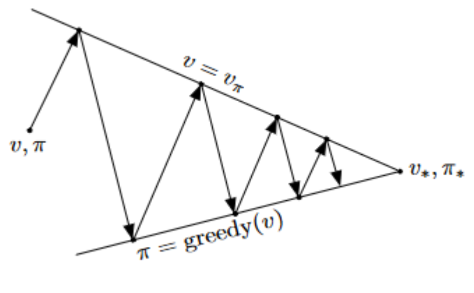


Figure 3.3: Generalised policy iteration visualised. Interleaving policy improvement and policy evaluation leads to eventual convergence to the optimal value-function and policy [1]

If at any point in the decision process some action a', not selected by the current policy π, results in a higher expected return value than the action selected by π, the current policy π is not the optimal one. A better policy π' is said to be found if the expected return of policy π from state s is lower than performing some other action a and thereafter following π. The policy improvement theorem, equation 3.12, states that a policy π' must be equal or better than π if:

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \tag{3.11}$$

Where $\pi'(s)$ represents the action a resulting from policy π'. To make this comparison, the Bellman value function equations would have to be known. Because these they are recursively defined, multiple passes (*sweeps*) over the state space may be required before they converge to their true value. Dynamic programming defines two general ideas for reaching this convergence: *policy iteration* and *value iteration*. Both policy iteration and value iteration improve the policy by acting greedily with respect to the current value function:

$$\pi'(s) = \arg\max_a Q^{\pi}(s, a) = \arg\max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^{\pi}(s')\right] \tag{3.12}$$

Where the relation between the action-value function $Q(s, a)$ and the state-value function $V(s)$ is:

$$Q^{\pi}(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^{\pi}(s')\right] \tag{3.13}$$

They differ in calculation of the value function. Policy iteration formulates the Bellman equation as an update rule, where the state-value becomes an accurate representation of the actual state-value of the current policy if the difference in value between consecutive updates becomes sufficiently small:

$$V^{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^k(s')\right] \tag{3.14}$$

Value iteration on the other hand uses the Bellman optimality equation (eq. 3.8) to effectively truncate the convergence after one iteration. Instead of performing multiple sweeps to get a better estimate of $V^{\pi}(s)$, the maximum value over action is taken as the nearest approximation of $V^{\pi}(s)$:

$$V^{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V^k(s')\right] \tag{3.15}$$

By iterating just once over the state value, value iteration effectively is a single policy evaluation and a single policy improvement. It is also possible to make a number of policy evaluations (eq. 3.14) before truncating it with equation 3.15, approximating the state value closer to its true value before maximizing over actions.

3.5. Learning from experience

While dynamic programming is a good strategy to find an optimal policy in an environment of which a perfect model is available, in reality this might not always be the case. Obtaining a model from a system can be too costly, the system might not be fully understood, or it is not fully known beforehand [31]. Reinforcement Learning (RL) considers algorithms that are able to learn policies from samples. Such samples can be collected in advance, or received on-line. Off-line RL methods apply when data can be obtained in advance, and on-line methods work through direct interaction with the system. Of course, when a system model is available, data can also be generated from there instead of from the real system. Indeed, this thesis work makes use of a simulated aircraft model [8] to resemble interaction with the actual system.

Learning from samples means learning from the tuple $[s, a, r, s']$ as perceived by the agent 'moving' through the state space. This limited set of knowledge about the environment prohibits a complete state-space sweep like in dynamic programming. Because of this property, exploration of the state-space is an important aspect to RL. With increasing exploration more knowledge about the MDP is gathered, and therefore a better policy can be learned. At the same time it is tempting to exploit what is already known to be good since this results in a high return. This trade-off is known as the exploration/exploitation dilemma [1]. There are also other problems that arise in reinforcement learning. In an environment where outcomes of behaviour are delayed in time, it is difficult to assign credit to the actions that contributed most to this outcome. This is known as the temporal credit assignment problem. The most important issue of RL this thesis aims to overcome is the *curse of dimensionality*: the exponential increase of computational requirements with the number of state variables [1]. Before diving into solutions for this problem, lets first consider some algorithms of RL.

3.5.1. Update methods

Similar to DP, reinforcement learning keeps track of the Bellman value functions to facilitate the learning process. The optimal policy results from the policy that returns the highest accumulated discounted reward. Since the state-value action is the maximum expected accumulated total reward from state s, always selecting an action resulting in the best state value is equal to the optimal policy (eq. 3.16). A practical way of describing this is by using the action-value function (eq. 3.17), the optimally formulated equivalent of eq. 3.12. In RL this function is also known as the Q-function. For each state-action pair it stores a *q-value*. An optimal policy would choose the action from the current state for which the q-value is maximized:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s) \quad \forall s \in S \tag{3.16}$$

$$= \arg\max_{a} Q^*(s,a) \quad \forall s \in S \tag{3.17}$$

Storage of these action-values is most simply done in a *tabular* fashion. The table size is the number of combinations between the set of states and the set of actions, the state-action pairs: $S \times A$. Considering discretized states and actions, is is the number of discretized states times the number of discretized actions. This is also why the size of the table grows exponentially with the number of state and action variables. Learning the Q-function is a matter of collecting and updating the q-values for each encountered state-action pair in the sample set. For this roughly two types of update strategies exist: the Monte Carlo (MC) method and Temporal Difference (TD) methods.

Monte Carlo methods assume an episodic learning environment. At the end of the episode the experience collected during the episode is used to update values and policies. A MC method can thus be incremental in episodes, but is not suited for online use [1]. MC learning is based on averaging sample *returns* for each state-action pair [1]. Return is different from reward since it is the discounted sum (eq. 3.2). TD methods on the other hand use experience during execution of the episode. Updates to the value functions and policy are carried out at every time step: TD(0). Because of this property, TD methods *bootstrap*. They back up estimates based partially on previous estimates. A blend of MC and TD, where MC backs values up only at the end of the episode and TD(0) at every step in the episode, is n-step TD. Here backups are made every n steps. To put the different learning methods in perspective, figure 3.4 shows the two dimensions of RL [32][1]:

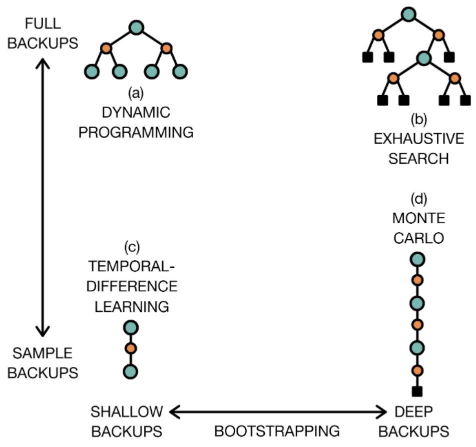


Figure 3.4: Dimensions of RL. At one end full backups are used to learn (a,b), while the other makes use of sample backups (c,d). Bootstrapping in TD-learning is performed every step and extends through n-step TD towards Monte Carlo methods not relying on bootstrapping. Figure from [32], an adaptation from [1].

For a Monte Carlo estimation, the root node of the backup is the episode starting state. In between are all states, actions, and rewards encountered on its trajectory, concluded by the terminal state. Monte Carlo methods only take the actual sampled transitions encountered during the episode, whereas DP takes all possible transitions from a state in account. Differences between DP, MC, and TD are mainly found in their approach to the prediction problem, estimation of the value function $V^{\pi}(s)$. For the control problem they all use a variation on the GPI (fig. 3.3) [1].

3.5.2. Algorithms

In reinforcement learning, temporal difference learning is one of the most important learning principles. TD learning can be implemented on-line, on episodic or continuing tasks, and in practise converges faster than MC methods [1]. The most popular algorithm is Q-learning [22]. Q-learning is an off-policy TD learning algorithm that updates the Q-function incrementally. For each encountered state-action pair the following update rule is applied [1]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s,a) \right] \tag{3.18}$$

Here the learning rate parameter α indicates by how much the current q-value entry should be changed based on the observed change after performing an action. Quantities like these (α, γ, number of episodes, etc.) are *hyperparameters*. Learning can be sensitive to these parameters, and setting them thus has to be considered carefully. Q-learning is *off-policy*, the acting policy can be different from the policy that is learned about. Instead of adding the discounted value of the action that would be taken according to the acting policy, Q-learning updates the q-value with the maximum expected return available. This is different to SARSA, an *on-policy* TD learning algorithm, where the update is performed using the action the acting policy would have chosen:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s,a) \right] \tag{3.19}$$

Although a policy is trained to point fingers at the current optimal action to take from a state, only taking this action would lead to exploitation only. To truly learn, the selection of actions mustn't always be the same as the optimal one. Therefore, several action selection strategies exist. Some of these are; ϵ-greedy, softmax action selection, and upper confidence bound (UCB) [1]. Instead of acting greedy all the time, ϵ-greedy selects with probability ϵ an action from the available set of actions with equal probability. Softmax action selection ranks actions according to the value they are expected to return. This value is normalized and turned into a probability of being chosen. Therefore the best action has the highest probability, and the worst the lowest. UCB bases selection on a measure of how often an action has been chosen to determine the uncertainty of knowing the true value of that action.

Methods like SARSA and Q-learning are *value-based*, they derive a policy directly from the estimated value function. There are also other algorithms that determine a policy by directly altering the policy itself; *policy-based*. A common classification of methods in RL is threefold; actor-only, critic-only, and actor-critic [33]. Actor-only methods search directly in the policy space, critic-only methods first find the optimal value-function and derive the optimal policy from it. Actor-critic methods combine both methods thereby containing both an actor and a critic. The critic learns the value function and the actor changes its policy based on that. Also, for each type of method it is possible to include a model in the learning procedure. Figure 3.5 displays the overlap and differences between these three classes [34]:

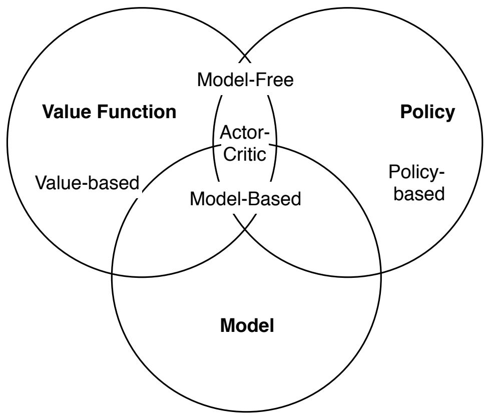


Figure 3.5: Taxonomy of learning strategies, adapted from [34]

Actor-only methods typically use parameters (θ) to represent the policy. Assuming the parameters are differentiable, gradient ascent can be applied w.r.t. a cost function to search for the optimal policy [35]. These algorithms are called *policy-gradient* methods. An early example of a policy gradient algorithm is REINFORCE [36], a Monte Carlo policy-gradient method.

Knowing in advance if an actor-only, critic-only, or actor-critic method will yield the best result for a problem is virtually impossible [35]. A survey about actor-critic RL [35] gives some guidelines: If the policy is required to produce actions in a continuous space, critic-only is not an option. They should be used in a small, finite action space. It then overcomes high-variance gradients in actor-only methods and spares introducing more hyperparameters. Actor-only methods are the preferred choice in a non-stationary environment, where the critic is too slow in adapting to the changing nature of the process. Actor-critic methods make the most sense in a (quasi-)stationary environment with continuous state and action space.

3.5.3. Large state spaces

Q-learning and SARSA discussed above require only a value function to be estimated. These are critic-only methods. These algorithms require an exact representation of the value function and policy to learn. This makes them inherently limited to small, discrete problems. Real life problems often have a large or infinite number of possible values or are continuous. Again, the combinatorial explosion of possibilities that have to be learned about for such problems is huge. In this case generalisation is needed. Actor-only method typically use a parametrized policy to do so, and actor-critic methods use approximation of both the policy and value-function. Approximation of the policy, value-function, and/or model enables a more compact formulation of the problem and reduces the number of values to be stored [31]. In approximate RL techniques of this kind are referred to as Function Approximation (FA) techniques, a supervised learning problem. Differences with normal supervised learning are potential nonstationarity, bootstrapping, and delayed targets [1]. FAs come in many different shapes and forms, can be linear, non-linear, gradient-based or gradient free [37]. Examples are; curve fitting, neural networks, and radial basis functions.

Within approximate RL three major classes can be distinguished; approximate value iteration, approximate policy iteration, and approximate policy search [31]. Each of those methods have an offline and online variant. Approximate value iteration algorithms search for the optimal value function. From this, the optimal policy is derived. Approximate policy iteration evaluates policies by constructing their value function, which in turn guides further improvement of the policy. Approximate policy search uses a direct policy search using optimization techniques (policy-gradient) [31]. There are however also some difficulties in approximate RL. Choosing an appropriate FA is a highly non-trivial task, and in general a more complex approximator requires more data to compute an accurate solution [31]. The maximization of the policy to find the action might be a problem in itself, and the estimation of the expected values from samples can be difficult [31].

Among the variety of FAs and classes mentioned, a revolutionary technique in reinforcement learning is deep learning. Deep learning enables scaling of reinforcement learning to problems previously intractable, such as learning to play video games directly from pixel information [38]. The powerful function approximation and representation learning properties of deep neural networks are a great tool in overcoming memory complexity, computational complexity, and sample complexity [38]. A major step in Deep Reinforcement Learning (DRL) has been the Deep Q-Network (DQN) [39]. It made learning with neural networks robust and stable, where it previously wasn't. The DQN agent surpassed performance of previous algorithms and achieved a level comparable to human players across many of the Atari games. The network, a deep convolutional network, uses hierarchical layers of tiled convolutional filters to exploit the local spatial correlations available in images to estimate the action-value function. After the DQN, another major step was masting the game of GO with AlphaGo [27], followed by the more general applicable AlphaZero also mastering chess and shogi and outperforming AlphaGo in Go [28].

Thus, several methods are available for scaling reinforcement learning to more complex and large problems. An additional idea is the restructuring of the learning framework from flat to multiple layers. The emerging hierarchical framework can facilitate splitting of the problem into multiple sub-problems on multiple temporal levels of abstraction. It makes planning and control on multiple time-scales possible, and can limit the search space for sub-problems. This approach is known as Hierarchical Reinforcement Learning (HRL), and will be the focus of chapter 4.

3.6. State-of-the-art in flight control with RL

Several researchers have put their efforts into application of RL in flight control design. The majority of recent articles use an Unmanned Aerial Vehicle (UAV) for application of their algorithms in the real world. With UAVs it is generally easier to conduct real-life experiments at a low cost and relatively low risk. This section will discuss relevant research by giving a short overview of it, thereafter discussing the relevance for this research. These discussions are somewhat dependent on the explanation given about HRL in chapter 4.

▷ *Online Adaptive Critic Flight Control* - S. Ferrari, R.F. Stengel [40]
In their article, Ferrari and Stengel discuss the design process of a flight controller for a simulated six Degree of Freedom (DoF) business jet over its full operating range. To solve the curse of dimensionality of classical DP, they use an Approximate Dynamic Programming (ADP) technique; the adaptive-critic design (also known as actor-critic). This design uses incremental optimization and parametric structures that approximate the optimal cost and control. It does not pose restrictions on the form of the dynamic equation or controller a priori. A critic is used to evaluate the performance of the actor. For both the critic and the actor a neural network representation was chosen since those easily handle large-dimensional space and can learn in batch or incremental mode.

Although ADP methods have shown to converge to the optimal policy over time, it is difficult to do so in real-time. Offline training of adaptive-critic controllers has been more successful. The paper therefore makes use of a two-phase training procedure, an offline phase, and an online phase. Size and weights of the networks are determined from earlier control knowledge in the offline phase. In the online phase these networks are updated (trained incrementally) to improve the control response, thereby accounting for differences between the actual and the assumed dynamic models. For the online phase a Dual Heuristic Programming (DHP) architecture was chosen to accelerate convergence to the optimal solution. The result is a flight control system that improves performance with respect to the offline specifications when subjected to unexpected conditions such as unmodelled dynamics, control failures, and parameter variations.

[Relevance to this thesis work] Ferrari and Stengel use a two-phase learning method that resembles how this thesis will attack the problem of flight control design. The first phase is meant to provide known information about the system to the second phase of the learning process. Similar to how the structure of the hierarchy will give information about the system before learning in HRL. Also, the system they opt to control is similar to the aircraft chosen for this thesis, a six DoF aircraft. For training, Ferrari and Stengel have resorted to ADP to cope with the large state space. This does however mean that the complexity of training increases w.r.t. the discrete exact representation case (e.g. Q-learning). Therefore it is important to establish to which degree the curse of dimensionality plays a role after application of a hierarchy. If so, the adaptive-critic design implemented in this paper can perhaps alleviate this problem in the online phase.

▷ *Autonomous Inverted Helicopter Flight via Reinforcement Learning* - A. Ng et al. [41]
The article by Ng et al. focuses on the design of a flight controller for the highly unstable low-speed flight region of a helicopter. Specifically, they design a controller for sustained inverted hover, an even less stable state than upright low-speed flight. First supervised learning was applied to identify a model of the dynamics in inverted flight from logged human pilot commands and helicopter states. This results in a stochastic Gaussian distributed model that gives a good estimate of the state transition function. The model is identified at 10Hz. Then, reinforcement learning is used on the model to search for a good controller policy. The designed reward function is chosen quadratic, where (x^*, y^*, z^*) and ω^* are desired position and orientation respectively. The coefficients α_i chosen to scale each term to roughly the same magnitude.

$$\begin{aligned} R(s^s) = &-(\alpha_x(x-x^*)^2 + \alpha_y(y-y^*)^2 + \alpha_z(z-z^*)^2 \\ &+ \alpha_{\dot{x}}\dot{x}^2 + \alpha_{\dot{y}}\dot{y}^2 + \alpha_{\dot{z}}z^2 + \alpha_\omega(\omega-\omega^*)^2) \end{aligned} \tag{3.20}$$

The controller policy is represented by a neural network. Training of the network is achieved in several steps. First Monte Carlo samples are taken from the model to get an approximation of the expected return. To solve this difficult stochastic optimization problem, the PEGASUS method was used to convert it to a deterministic optimization problem. With it, a greedy hillclimbing algorithm was used to search for a policy with high estimated approximate return. The resulting controller is successful in sustaining inverted flight, and also capable of normal upright flight, including hover and waypoint following.

[Relevance to this thesis work] The paper by A. Ng et al. shows the possibility of achieving more than adequate control over a remote controlled helicopter using approximate reinforcement learning with a neural network as the policy. A main advantage that was found is the quick adaptation of the control policy after reconfiguration of the helicopter between flights. This shows the adaptability of these algorithms, also useful for fixed-wing aircraft. Also interesting is the formulation of the reward function. Since the aircraft will have to adhere to a given trajectory as well (position and attitude), a similar reward function seems a viable option. The design procedure taken in the paper, however, does complicate the search for a good policy since multiple methods had to be used. Perhaps by using a hierarchy simple Q-learning can be utilised instead.

▷ *Self-tuning gains of a quadrotor using simple model for RL* - J. Junell et al. [42]
Junell explores the possibility of controller design for a quadrotor without the use of function approximators to deal with the high dimension of the problem. Instead, the learning task is simplified by first learning on a simple or incomplete model. By using this hybrid approach to do some of the heavy computational work, the number of real-life learning trails is decreased whilst retaining convergence guarantees to a locally optimal policy. This approach is therefore advantageous to vehicles that need local tuning of gains. The optimization algorithm uses a flight control structure in which it is supposed to learn set the appropriate gains. Starting from an initial hand-designed gain set, a policy search is applied to find locally optimal gains. The algorithm is tested on a simulated F-16 model and a simulated quadcopter. Both simulations showed an increase in performance over each iteration on the complex system variant. It was noted that the initial gain set will highly influence the performance of the final policy since it is only optimizing locally.

[Relevance to this thesis work] The research shows the use of reinforcement learning in a context where existing flight controller structures are re-used. Instead of viewing the problem as a flat one, controller gain tuning is the objective of the learning algorithm. This research will in a similar way attempt to include knowledge of this type in the learning process to simplify the process. An important result from this research therefore is that it was found possible to train a policy using an incomplete or simplified model first. This shows that inclusion of prior knowledge of this type helps greatly in reducing learning effort afterwards. On top of re-using controller structures this might therefore prove to be an additional method to reduce learning effort for the hierarchical reinforcement learning application in this thesis.

4 Hierarchical Reinforcement Learning

Reinforcement learning is a great paradigm to find suitable control strategies for relatively small problems. Scaling reinforcement learning to larger problems will however reveal some limiting factors. Basic algorithms of RL treat the state space as one huge flat search (flat-RL) [2]. Due to the exponential growth of the to be learned parameters with the number of state variables, the curse of dimensionality, this method is incredibly sample-inefficient, time-consuming or plain infeasible. In flat-RL, actions are required at each instant in time, thereby leaving behind the ability to take advantage of simplicities and efficiencies sometimes available at higher levels of temporal abstraction [4]. Temporal abstraction refers to the use of 'high-level' actions with varying durations, in contrast to the 'primitive' or 'low-level' actions in the standard MDP framework of flat-RL [25]. Some examples of temporally-extended activities are; temporally-extended actions, macro-operators, sub-tasks, modes and behaviours. This report will refer to them in a general sense as 'activities', as in [5]. Activities can be triggered to follow their own policy until termination [5]. They are generally defined on part of the state space only [5], and are therefore smaller problems. The use of activities splits the search space into multiple sub-problems, instead of treating it as a single space. This naturally leads to a hierarchical structure. By solving each of the sub-problems, the total problem is solved. The group of algorithms that use a hierarchical organisation is known as Hierarchical Reinforcement Learning (HRL).

The hierarchical structuring of a learning process seems only natural when observing human behaviour in this context. In our daily behaviour we include certain sub-routines. The navigation from home to work is logically subdivided into navigations from intersection to intersection. The navigation indoors is performed from door to door until the desired room is reached. There are plenty of examples where we order information on different levels of temporal abstraction, and seldom is the task seen as one single task. Cognitive psychologists distinguish two types of knowledge: declarative- and procedural knowledge [43]. Declarative knowledge is 'knowing what', procedural is 'knowing how' [43]. HRL communicates declarative information through the structure of the hierarchy. Within that structure the subtleties (know-how) of the exact control commands will be learned. Differently put, it is easier to learn about activities than it is to learn about a giant sequence of primitive actions. Also, HRL allows for the inclusion of prior knowledge about as well as for re-usability of activities in other parts of the problem domain. A suitable property for the implementation of existing flight control techniques. All mentioned properties make HRL an important method for battling the curse of dimensionality. It has to be noted that badly formulated, or unhelpful activities can increase the difficulty of the problem [44] and might introduce a slight sub-optimality in performance compared to flat-RL [45]. So while care has to be taken in designing the hierarchy, it can provide a fundamental improvement in learning over flat-RL. This is the main motivation for the exploration of a hierarchical reinforcement learning approach for flight control.

Construction and shaping of this hierarchical learning environment can be achieved in a variety of ways. This chapter discusses the main theoretical formulations of HRL, and explore ways to fit the flight control structures discovered in chapter 2 within these. First the theoretical elements of HRL are discussed in section 4.1. Section 4.2 discusses three main approaches to HRL; options, Hierarchies of Abstract Machines (HAM), and MAXQ respectively. Additionally Feudal-RL is discussed shortly. Thereafter the state-of-the-art in HRL is discussed in section 4.3. Finally the combination of flight control structures and HRL techniques is discussed.

4.1. Elements of HRL

The decision process of a flat reinforcement learning problem is modelled as a Markov Decision Process (MDP). While this process encapsulates the sequential process, it does not account for the time that passes between chosen actions [5]. For treating problems with a temporal abstraction the theory of the Semi-Markov Decision Process (SMDP) [46] is key [4]. The SMDP models temporally-extended activities where actions take variable amounts of time [4]. The discrete-time SMDP formulation underlies most approaches to HRL [47]. Both Q-learning and SARSA apply also to SMDPs, if the immediate reward is interpreted as the return accumulated during the execution of a temporally-extended action, discounted properly. Equation 4.1 shows Q-learning for the SMDP case, where τ is the time it takes to transition to the next state [5]:

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k \left[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \gamma^{\tau} \max_{a' \in A'_s} Q_k(s',a') \right] \tag{4.1}$$

While the theory of SMDP specifies how to model and plan these types of problems, the structure in which the different levels of hierarchy are placed determines the relation between higher-level activities and low-level actions. Two elements of the hierarchy are of importance; decomposition and structure. The decomposition into smaller activities determines the 'working area' for each of the activities, therefore also called a partial policy. The structure in which they are linked together determines how they interact and relate to each other. This decomposition and structuring is the subject of different formulations in the HRL framework. Their goal is to provide guidance for the structuring, decomposition, and interrelation of a problem, such that an appropriate hierarchy of activities can be made. The three main formulations in HRL are options, HAM, and MAXQ. Irrespective of the choice of hierarchy, learning in HRL is performed in the same way as learning in flat-RL. Instead of just one policy, there are multiple ones to be trained. A high-level policy learns to activate the appropriate lower-level policies, and the lowest-level policies learn to generate primitive actions. The highest level policy for example can learn to choose among its available activities in the same way a primitive controller would learn using e.g. Q-learning. An example hierarchy demonstrating the division into different layers is shown in figure 4.1. All hierarchy is part of the agent, where from the highest level actions are chosen until at the lowest level finally primitive actions are generated. It is possible that a structure defines overlap between different higher-level agents. An example is that they can select the same lowest level primitive action policy, but base their selection differently on their own policy. Also at multiple levels in the hierarchy it can be possible to select a primitive action.

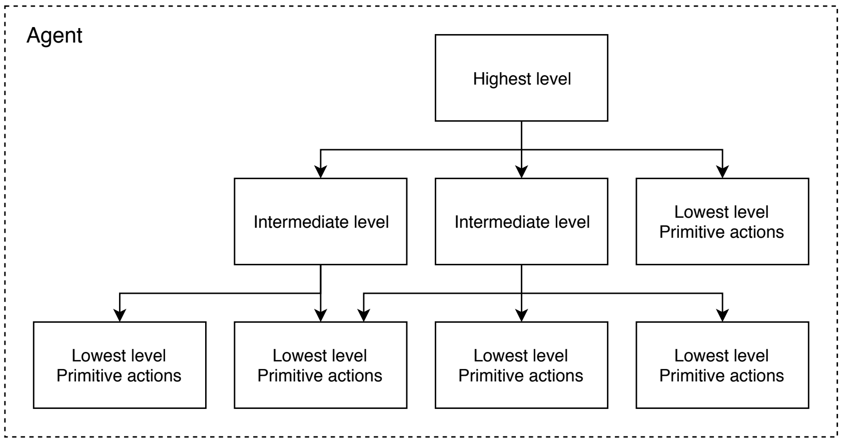

Agent
Highest level
Intermediate level
Intermediate level
Lowest level Primitive actions
Lowest level Primitive actions
Lowest level Primitive actions
Lowest level Primitive actions
Lowest level Primitive actions

Figure 4.1: An example hierarchical structure

An interesting idea is the implementation of activities that do not require learning, but are simply already made to perform a certain task using e.g. classical control. Choosing this route can help move the learning process away from trying to learn known optimal solutions, and rather focus on the aspect of when to select this solution. An example is the use of a pre-tuned Proportional-Integral-Derivative (PID) controller, that is selected whenever a certain desired roll-rate needs to be achieved. Learning the correct aileron motions would result in spending learning cycles on a task of which it is known that PID control already performs well. Of course the PID controller can be made adaptive such that the controller gains can be set by a higher level policy as well. Including these types of ideas in the hierarchy can greatly reduce the effort spent on learning the appropriate actions for a higher level goal, further reducing the effect of dimensionality.

4.2. Main Hierarchical approaches

Hierarchical reinforcement learning emerged as a potential solution to the curse of dimensionality found in flat-RL. The earliest and still most popular hierarchical approaches are Hierarchies of Abstract Machines (HAM) by Parr & Russell [25], options by Sutton [4], and MAXQ by Dietterich [2], in order of discovery. Although variants exist, these three are the most popular forms of HRL. This report also mentions Feudal RL by Dayan [48] because it may provide useful in learning appropriate navigation goals for flight control.

Among the methods mentioned, the concept of 'activity' varies. Options view them as options to choose from, HAM defines a multitude of machines that manage action generation, and MAXQ sees them as subtasks linking together in a tree. Feudal RL defines managers that operate each at a more granular part of the state space. This section will shortly go over the framework each of the approaches impose on the problem.

4.2.1. Options

Even though HAM and feudal RL were formulated before the others, we will consider options first since it is closest to the MDP formulation of flat-RL. In that MDP the repeating sequence of state, action and reward, including the decisions at each instant in time are modelled. An SMDP requires decisions only at larger time intervals. One way to visualise this is figure 4.2 [4]. Here the agent follows the same path through the state space as the MDP but its primitive actions are dictated by the choice of a higher-level policy selecting sub-policies at each decision point, rather than having one policy select actions at each step.

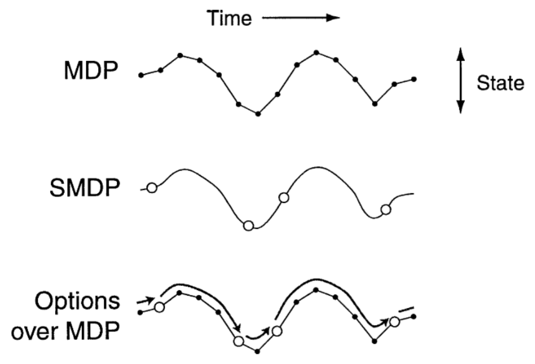


Figure 4.2: State-transition structure of options [4]

The available set of policies at each decision point in the SMDP chain are known as options. An option is not a sequence of actions, but a closed-loop control rule [49]. It consists of an initiation set, a policy, and a termination condition: $< I, \pi, \beta >$. The initiation set and termination condition restrict the range of application of the option to within certain bounds, implying a limitation to the range over which a policy needs to be defined [4]. The initiation set is a subset of the state space. The termination condition can hold anything indicating a target has been reached, or the policy is taken outside of its domain of application [4]. This termination condition can also include a period of elapsed time or be dependent on the history (Ω) experienced by the agent, in contrast to a normal MDP.

The policy in charge of selecting options is the policy over options. With only one such policy, the hierarchy has a depth of one. In a deeper hierarchy, options are able to select options. Only the lowest-level option selects primitive actions, generating transitions and reward. This is known as the *one-step* option. All other options are then regarded as *multi-step* options. A hierarchy of options can form a structure as in figure 4.1. Upon termination of option o at state s, the accumulated discounted reward of that option is used to update the higher-level option that initiated its execution. Updating the value functions is performed using traditional RL techniques. An example is the use of Q-learning, where the higher-level policy/option is updated with the accumulated reward from the option it initiated:

$$Q_{k+1}(s,o) = (1-\alpha_k)Q_k(s,o) + \alpha_k \left[r + \gamma^{\tau} \max_{o' \in O'_s} Q_k(s',o') \right] \tag{4.2}$$

By performing an update rule like Q-learning, the hierarchy will increasingly understand what options are beneficial to perform from what state. However, this update rule will only trigger as the option terminates. A variant is *intra-option* learning, that uses information gained during execution of the option for learning [50].

There are several advantages to the use of options for learning over flat-RL. The formulation of options is closest to action selection in flat-RL, so planning and learning as performed in flat-RL remains the same [5]. The use of options is also beneficial in the initial stages of learning, where options prevent a period of 'flailing' often seen in RL systems [5]. In addition, and similar to other hierarchical approaches, options facilitate the transfer of learning knowledge to similar tasks. But this depends entirely on the decomposition of the problem and the task of the option. The focus of options as a method is to augment the core MDP with temporal extensions [4]. However, if the set of options does not include the one-step options corresponding to the entire set of primitive actions, options simplify rather than augment [5]. Simplification is the focus of HAMs and MAXQ.

4.2.2. Hierarchies of abstract machines (HAM)

Different from options, the focus of Hierarchies of Abstract Machines (HAM) is to simplify the MDP rather than augment by restricting the class of realizable policies rather than expanding action choices [5]. A hierarchy of machines is defined as a collection of deterministic finite state machines that have the ability to call each other until a primitive action is reached in the core MDP. Important features of this approach are; an agent-centered abstraction, hierarchical use of prior knowledge, the optimal refinement of incompletely specified behaviours, and state space reduction [25]. Again the parallel to human behaviour can be drawn. Often humans have generic rules for situations that can be adapted quickly to suit a specific task. HAMs tries to encapsulate this behaviour by using prior information and then learning when or why to specifically apply this knowledge. The focus is not on primitive control operations, but rather at higher level intelligent decision making since low-level feedback control should be handled using control theory techniques or highly specialised **AI!** (**AI!**) techniques [25].

The HAM approach views policies as *programs* that produce actions based on the agents information. A HAM policy consists of a collection of stochastic finite-state machines. Each machine individually can be in one of either four states; action, call, choice or stop. A machine is hierarchical if it calls other machines as a sub-routine [25]. A brief description of the states is given in [5]: Actions generate an action on the core MDP. A call state halts execution of the current machine, and starts execution of the called machine. A stop state terminates execution of the current machine, and returns control to the machine that called it. A choice state non-deterministically selects the next machine state. Actions eventually produced in the hierarchy result in a transition and reward in the core MDP. A state-transition structure of HAMs is given in figure 4.3 [25].

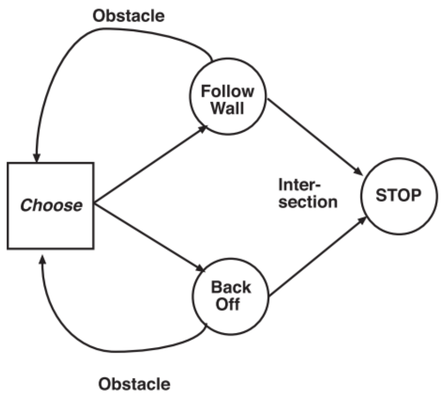


Figure 4.3: Hierarchies of abstract machines state-transition structure [25]

This machine resembles a robot in a navigation task, where it can either follow the wall or back-off after reaching an obstacle. After detecting it has reached an intersection the machine terminates. Relevant choices for a learning algorithms take place only in the choice state of the machine. By learning the appropriate selection probabilities for executing each of the actions, the agent improves. The Q-learning update algorithm for SMDPs given in equation 4.1 can thus be applied between each choice point [5]:

$$Q_{k+1}([s_c, m_c], a_c) = (1-\alpha_k)Q_k([s_c, m_c], a_c) + \alpha_k \left[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{\tau-1} r_{t+\tau} + \gamma^{\tau} \max_{a'} Q_k([s'_c, m'_c], a') \right] \quad (4.3)$$

Where τ is the time since the last choice point, s_c the current machine state, m_c the choice state, and a_c the action selected at the choice point. After convergence to Q^*, the optimal policy is found by greedily selecting choices with respect to it.

4.2.3. MAXQ Value function decomposition

MAXQ takes the approach of decomposing the value function of the core MDP into an additive combination of value functions of smaller MDPs [2]. It is based on the assumption that the designer is able to set useful sub-goals and define subtasks that achieve these goals, thereby constraining the set of policies that need to be considered [2]. MAXQ therefore simplifies the original problem. This structure allows for subtasks to be defined on a part of the state space only, thereby exploiting state abstraction. The decomposition makes sharing of the value function of sub-problems and re-use of the learned policies possible. Also, with state abstraction the total representation of the value function becomes smaller and will be faster to learn [2]. Unlike options and HAMs, MAXQ doesn't define a single SMDP, but rather creates a hierarchy of multiple SMDPs, learning them simultaneously [5]. Each subtask of MAXQ has three components; a policy (only active on part of the core MDP), termination condition, and a pseudo-reward function. Primitive actions are generated at the bottom of the task-tree. A hierarchical policy is a set containing the policies for each of the subtasks in the problem: $\pi = \pi_0, \cdots, \pi_n$ [2]. The tasks are related to each other via a task-graph, figure 4.4 [2]:

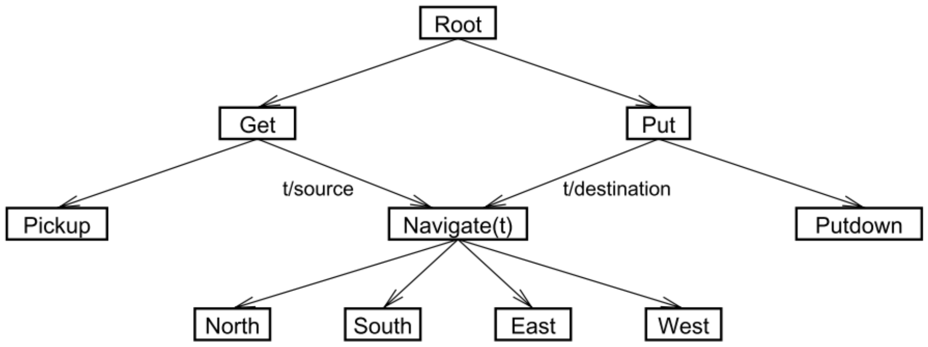


Figure 4.4: MAXQ state transition structure [2]

This particular task graph represents the problem of moving passengers around with a taxi. Both getting and dropping off passengers share the commonality of having to navigate. So depending on having reached the goal or not the taxi will in either case have to move in one of four directions. Starting at the root, MAXQ keeps track of the called subroutines (policies) in the tree through a pushdown stack K. At any time step the top of the stack contains the name of the subtask that is currently being executed [5]. The hierarchical value function then gives the expected cumulative reward following policy π from state s_t with stack K_t [2]. Hence, a hierarchical policy in MAXQ implicitly defines a mapping from state s_t and current stack K_t to primitive action a. The action-value function is a function of the current subtask i, state and action:

$$Q^\pi(i,s,a) = V^\pi(a,s) + \sum_{s',\tau} P_i^\pi(s',\tau|s,a)\gamma^\tau Q^\pi(i,s',\pi(s')) \tag{4.4}$$

Where the latter part of the equation is called the completion function (C^π). To find the value of the state at the root of the tree, a summation of values over the tree is taken. This is the value of s in the core MDP, the root value function:

$$V^\pi(0,s) = V^\pi(a_n,s) + C^\pi(a_{n-1},s,a_n) + \cdots + C^\pi(a_1,s,a_2) + C^\pi(0,s,a_1) \tag{4.5}$$

By updating this value recursively, the optimal policy can be learned. The MAXQ-Q algorithm uses Q-learning to update the completion functions. Maximizing the value function leads to the optimal policy.

4.2.4. Fuedal HRL

A fourth hierarchical form in HRL is feudal RL [48]. The name originates from the feudal structure in the middle ages, where the head of society determines desired goals and the workers simply adhere to that wish. Like MAXQ it defines a task hierarchy that couples different layers to one another. Different to MAXQ, there is no decomposition of the value function. Rather, each level has its own Q-function on which decisions are based. It can be compared to the managerial layers in a large company. Sub-managers do not need an understanding initially of their managers' commands, they simply maximize their reinforcement within the current command [48]. In this structure, the top level managers reason at a lower temporal resolution and set out long-term goals. The lowest level managers perform primitive actions and have the finest granularity in time. Two principles are key in the feudal structure; reward hiding and information hiding. Managers reward sub-managers for reaching the goal set, regardless of whether it satisfies the managers own goal. This learns sub-managers to achieve goals even if the manager set the wrong goal. Also, managers only need to know the system state at their own level of granularity of choices. Information about the higher level task, or lower level actions are hidden from the current level. The task hierarchy is especially useful for different layers of temporal abstraction. The example given in the original paper is a navigation task. Figure 4.5 shows four levels of a fuedal hierarchy. The highest level manager determines a direction only, and lower level managers specify more precisely which coordinate is desired to achieve. In this case a U-shaped barrier needs to be avoided.

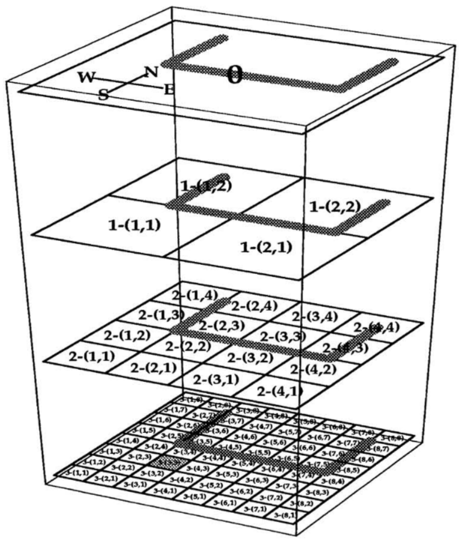


Figure 4.5: Feudal hierarchy granularities. Each level is divided into four lower levels [48]

A fifth possible action (*) denotes that the lower level manager should search for a goal within the current higher level space, instead of moving to another space. At the top level the only action therefore is *, while the lowest level can only execute N-S-E-W movement. Feudal RL is slow to converge in the initial learning phase, but surpasses flat-RL in later stages [48]. Initially exploration produces impossible actions, but later on the higher level capacity provides a significant advantage.

4.2.5. Deep Hierarchical Reinforcement Learning

Having mentioned the basic forms of HRL; options, MAXQ, HAMs and Feudal RL, there are some recent developments that combine these ideas with deep learning to form deep hierarchical reinforcement learning. Three forms are the hierarchical DQN (h-DQN), option-critic, and Feudal Networks (FuN).

The hierarchical DQN [51] combines the DQN technique with a hierarchy. The focus is on sparse environments and employment of intrinsically motived agents that strive to improve behaviour for their own interest rather than to solve external goals. The h-DQN framwork defines a top-level q-value function to learn a policy over options, with the lower level learning a policy to accomplish these options. The model is trained using stochastic gradient descent at the different temporal scales to optimize expected future extrinsic (high level) and intrinsic (low level) rewards. This approach proves strong on problems with sparse rewards where most existing state-of-the-art deep RL approaches fail to learn policies in a data efficient manner [51].

The option-critic architecture [52] is an extension of the options framework. The article by Bacon et al. focuses on the aspect of automatically creating temporal abstractions from data. Where options are normally defined before learning commences, discovering temporal abstraction autonomously has been the subject of extensive research efforts. The majority of work has gone to finding subgoals, and learning policies to reach those [52]. The option-critic architecture is a gradient-based approach capable of learning option policies as well as option termination conditions, simultaneously with the policy over options. Only the desired number of learned policies needs to be specified.

Feudal networks [53] is an extension of Feudal RL. Just two levels of hierarchy are defined, a Manager and a Worker. As with Feudal RL, the manager works on a lower temporal resolution and sets goals for the worker. The worker generates primitive actions at every tick. It encourages the emergence of sub-policies for different goals of the manager. The network architecture is a fully differentiable neural network with two hierarchy levels. The manager sets goals for the worker though a latent space representation learnt by the manager. The worker has an intrinsic reward to encourage following the goals set. FuNs shows that it makes long-term credit assignment and memorisation more tractable.

4.3. State of the art applications of HRL

Having discussed the main hierarchical reinforcement learning appearances and recent state-of-the-art deep HRL forms, this section will go over state-of-the-art implementations of HRL. In the field of flight control these are scarce. To the best of knowledge HRL has not been applied to flight control of a fixed-wing aircraft as proposed by this study. However, several other research on problems of similar complexity have shown the advantage of HRL over flat-RL. In a robot navigation task the addition of deliberate control in addition to just acting reactive with respect to the environment proves beneficial [54]. The advantages of are present in multiple levels of temporal abstraction. Simply having a goal to work towards in the not-to-distant future is a basic principle of many reasoning in human cognition. Recent research in HRL is, similar to the state-of-the-art in RL, focussed on the combination with deep learning, deep HRL. This section discusses some of this research in a similar manner to chapter 3.

▷ *Hierarchical Reinforcement Learning with Hindsight* - A. Levy, R. Platt, K. Saenko [55]
When rewards are spare and delayed, RL suffers from poor sample efficiency. Whilst existing work in HRL often assumes expert demonstrations or automatic discovery of temporal abstraction, they typically do not outperform flat-RL initially. The approach by Levy et al. combines universal value functions with hindsight learning, to learn policies in different time scales in parallel. To do so, goal states are introduced altering the MDP into a universal MDP (UMDP). With hindsight experience replay, the agent replays actions with different goals. Also, agents replay higher-level decisions using subgoal states achieved in hindsight as the subgoal actions. This allows the agent to evaluate higher level action even when lower levels have not fully learned to achieve these goals. This improves sample efficiency greatly. The method showed to significantly improve sample efficiency in discrete and continuous tasks.

[Relevance to this thesis work] Since many levels of temporal abstraction exist in the flight problem, as explained in chapter 2, learning in parallel among them can greatly help. Because improving the sample efficiency is one of the objectives of this research this method can prove to be a viable supplementary technique on top of introducing a hierarchy.

▷ *The Actor-Judge Method: Safe State Exploration for Hierarchical Reinforcement Learning Controllers* - S. Verbist, T. Mannucci, E. van Kampen [56]
Verbist uses hierarchical reinforcement learning to focus on the safety of exploration in RL. Inherent to exploration is the trail-and-error procedure leading to entering of unsafe states, very relevant for the control of physical systems. Without the assumption of full knowledge of unsafe states, the research proposes the actor-judge algorithm. This defines a hierarchy of high-level strategists and low-level tacticians where strategists plan to reach the goal set by the root node. Tacticians perform primitive actions. The difference with rewards in a normal RL setting is that the agent upon receiving a negative reward though collision might falsely perceive that it hasn't hit the wall hard enough. The judge is in place to prohibit the agent from exploring further, thereby constraining exploration to useful states. It has the possibility to override the agents proposed action. The method is tested on a quadrotor on 2D navigation task. It shows that HRL in itself is already an approach to safe reinforcement learning because the to be explored state space is smaller. The actor-judge proves a safer method for exploring, but normal HRL outperformed it at the cost of collisions.

[Relevance to this thesis work] Although the safety aspect of training is not of immediate concern to this thesis, the proposed method is interesting. It shows the possibility of softly guiding the learning process away from undesired states. This allows to set boundaries for learning aircraft motions with the goal of restricting exploration from definite undesired states. For example inverted flight or upright aircraft motion can then be eliminated other than by giving these states very negative rewards. Perhaps when the state space is larger than the problem considered by Verbist, this method might actually prove beneficial to the learning process.

▷ *Towards learning hierarchical skills for multi-phase manipulation tasks* - O. Kroemer et al. [57]

Kroemer et al. recognises that a most manipulation tasks are composed of a sequence of phases, where the actions have different effects in each phase. The robot can transition between phases thereby altering the effect of its actions. These transition points therefore are important subgoals to the overall task. This research presents a method for learning manipulation skills based on the phase structure of the task. The robot learns motor primitives for each of the phases optimizing it using a policy search. A higher-level policy uses a value function approach to learn sequencing motor primitives. Therefore the robot can learn policies that reuse motor primitives between tasks that have the same phase transitions. A human demonstration is used to learn a model of the phases and their transitions.

[Relevance to this thesis work] The consideration that complicated motor activities are linked to each other through phase transitions is an important realisation. It leads to a hierarchical form of learning where each level can be trained independently and is smaller in size. This method bears similarity to the motion of an aircraft, where also several phases exist in the motion patterns. While the research is focused on learning the phase transitions, it does show that having such a structure leads to a improvement over flat learning. Also the method was quite flexible in that it could adapt to different shapes of objects. Similar to how not every aircraft motion is exactly the same. Without the inclusion of the automatic phase discovery, this method closely resembles the option framework. So by having well defined options the learning process can be made more efficient and adaptive to slight differences.

5 Preliminary Analysis

To put theory into practise, the discussed reinforcement learning and hierarchical reinforcement learning techniques have been compared in a preliminary analysis. As a test-bed an inverted pendulum is used. This pendulum, shown in figure 5.1, is an under-actuated system where the goal is to learn to balance the pendulum vertically upright through application of various magnitudes of torque. The problem is complicated by the under-actuation effectively splitting the problem into two phases: swing-up and balance. Gaining enough momentum to overcome the lack of immediate torque available and reach the desired upright attitude is part of the swing-up phase. This phase typically requires large torque input for most efficient use of time. Staying upright thereafter requires smaller torque input and is simply a matter of minimizing motion.

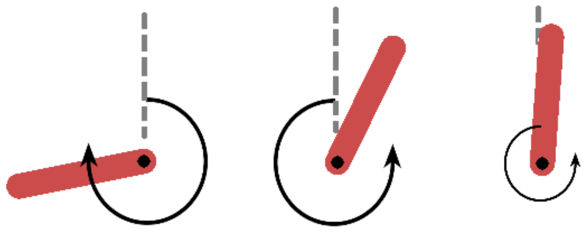


Figure 5.1: Inverted pendulum: A sequence of motions leading to balancing in vertical upright orientation

The inverted pendulum environment is courtesy of the OpenAI Gym environment collection [58]. This gym is a toolkit for development and comparison of reinforcement learning algorithms [58]. Different environments are available for training. These include reward signals and visualisations as well. The inverted pendulum was chosen for this preliminary analysis since it exhibits dynamic behaviour and induces delayed reward behaviour. Simply applying continuous torque will initially lead to improved reward collection, but only trading off these immediate rewards for increased momentum to reach higher attitudes ($\rightarrow \theta = 0$) will obtain true optimal reward collection. The pendulum consists of two states and one action variable; the angle, angular velocity, and applied torque. Limits of each variable are shown in table 5.1

Table 5.1: Inverted pendulum characteristics: limits of states and actions

Variable	Symbol	Min	Max	Unit
Angle	θ	$-\pi$	π	rad
Velocity	$\dot{\theta}$	-8	8	rad/s
Torque	T	-2	2	Nm

As a convention, $\theta = 0$ will be regarded as the angle in which the pendulum is hanging downwards when plotted. The reward function of the environment is given as in equation 5.1, putting most emphasis on obtaining the upright vertical orientation, thereafter maintaining low angular velocity, and thereafter minimizing torque application. Formulated as a negative signal, discouraging lingering.

$$r_t = -(\theta^2 + 0.1 * \dot{\theta}^2 + 0.001 * T^2) \tag{5.1}$$

An environment has a maximum execution duration of 200 time-steps. It can be loaded into simulation by using a specific or random seed. Seeds correspond to a combination of initial angle and velocity. The initial angle can be anywhere on the circle, and the initial angular velocity is between 0 and 1. For training purposes, a separate training- and test-set are defined. Their seeds are chosen such to not overlap with each other, and include only angle initialisations located in the lower half of the circle. This forces each initialisation to require the swing-up motion to be present in the solution, since from that situation balancing alone does not suffice. The test-set has a length of 200 individual seeds, and is used to compare algorithms to each other. This benchmark test determines the average reward received over 200 seeds with the method.

To demonstrate the differences between flat-RL and HRL with a focus on sample efficiency, a Q-learning algorithm is applied. State and action spaces are therefore discretised into a number of bins. To accommodate the full range of each variable, the discretisation forms equidistant bins. An example discretisation space of five for each state and action variable is given below:

$$d\theta = [-\pi, \quad -0.5\pi, \quad 0.0, \quad 0.5\pi, \quad \pi] \tag{5.2}$$

$$d\dot{\theta} = [-8.0, \quad -4.0, \quad 0.0, \quad 4.0, \quad 8.0] \tag{5.3}$$

$$dT = [-2.0, \quad -1.0, \quad 0.0, \quad 1.0, \quad 2.0] \tag{5.4}$$

Where the bins of e.g. $\dot{\theta}$ are spaced as:

$$\dot{\theta}_{bins} = [-8.0, -4.8], \quad [-4.8, -1.6], \quad [-1.6, 1.6], \quad [1.6, 4.8], \quad [4.8, 8.0] \tag{5.5}$$

Thereby thus regarding values between -8.0 and -4.8 as a single value. Within this range the agent is not able to distinguish differences. This is important for the precision of the information that is fed to the agent. Too coarse of a discretisation space leads to sub-optimal or no desirable results at all. Specifically for this problem is it important to have a fine enough state-space grid to (more precisely) detect the turn-over point in the swing-up phase at which the pendulum's rotation needs to be flipped to gain momentum. On the other hand, too fine of a discretisation complicates learning with RL due to the curse of dimensionality.

5.1. PID Control

To somewhat formalise the problem, consider figure 5.2 where a simple classical control solution to the problem using PID control is shown:

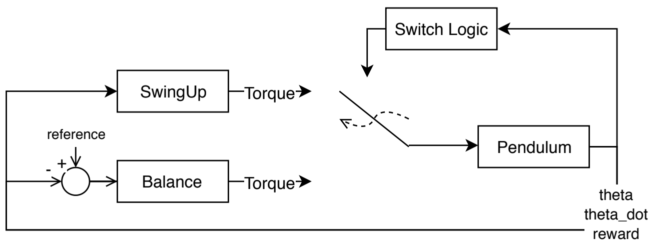


Figure 5.2: Inverted pendulum control scheme using PID

The figure shows a closed loop control system where the actions to the pendulum are either generated by a swing-up controller or a balance controller. Control switch logic determines which output is sent to the pendulum. No use is made of the reward signal. Both controllers are tailored specifically to one of two phases in the inversion process:

1. Swing-up: Apply max torque if velocity is positive. Apply min torque if velocity is negative.
2. Balance: Apply PD regulated torque control to track a reference (θ=0, $\dot{\theta}$ = 0)

The balance controller is manually tuned to adequate performance with settings: P = 10, D = 2. The swing-up phase is set active by default. The switch logic determines when the controller output is swapped to the balancing act. This is where the remaining logic is housed. All of these settings were manually tuned, and the following logical conditions are imposed on the switch:

1. $|\theta| < \pi/4 \quad$ OR $\quad |\dot{\theta}| > 5 \quad \rightarrow$ Switch to balance
2. $\dot{\theta} > 6 \quad \rightarrow T = 0$

Although these values can potentially be improved through refinement, their performance is satisfactory. A time trace of the angle, velocity, and torque input of this solution is shown in figure 5.3. For comparison each time trace shown in this chapter is performed using the same initial conditions (seed 0) of the environment. Angles in the plot are plotted on the domain $[-\pi, \pi]$, where 0 indicates the pendulum hanging down. This representation was chosen to visualize the swing-up procedure of the pendulum.

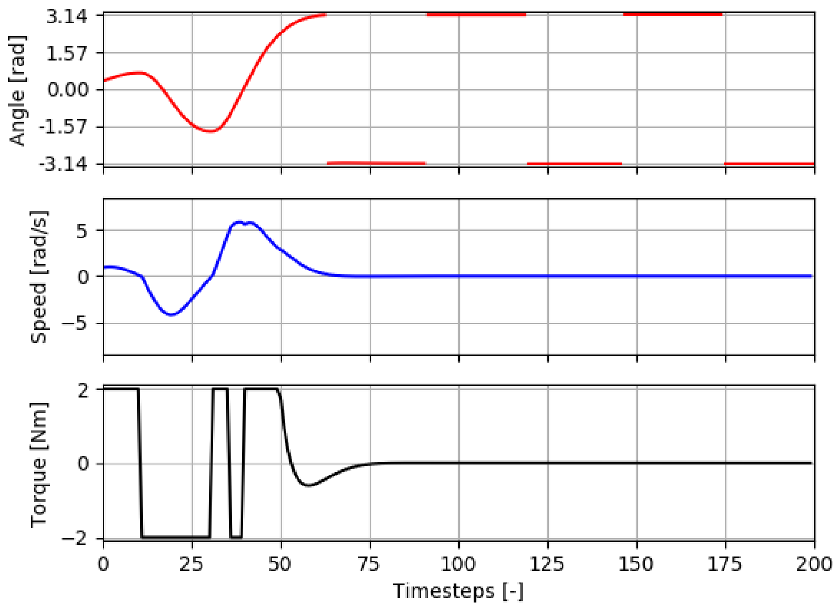


Figure 5.3: PID time trace showing angle, velocity, and torque curves over 200 time-steps of the simulated environment

Clearly visible in the time traces is the swing-up controller being active during the first ~50 time-steps. Around that time the switch to the balance controller is made, after which a typical PD torque-curve is visible. A slight oscillation around the unstable top equilibrium point is visible. The classical PD control solution scores -265 on the benchmark test, the average performance on 200 different initialisations of the problem.

5.2. Flat Reinforcement Learning

To see how reinforcement learning can be used to achieve a solution to the inverted pendulum, first a flat-RL approach is used. Here, as explained in chapter 3, the search space is seen as one huge space, and the problem is directly solved using estimations of the value of state-action pairs. For each training performed hereafter, including training of HRL, the hyperparameters shown in table 5.2 are in effect, and have been found empirically. The number of angle and velocity discretisations have found to be sufficient and not overly discretised with 25 bins. As discussed later in this section, the number of action discretisations is best set at four. Although this section will explore the effect of more discretisations, the remainder of this chapter will have four actions as the default setting. The number of episodes is calculated separately.

Table 5.2: Hyperparameters for training in reinforcement learning

Variable	Value	Explanation
spsap	400	samples per state-action pair
γ	0.95	Discount rate
α	0.10	Learning rate
ϵ_{min}	0.20	Minimum greedy ratio
ϵ_{max}	0.90	Maximum greedy ratio
dθ	25	Angle discretisations
d$\dot{\theta}$	25	Velocity discretisations
dT	4	Torque discretisations

On the topic of finesse in control indeed the number of discretisations is of importance to the control precision of the solution, but also to the time is takes to find a solution. Therefore a test analysing the effect of reduction of the search space on the benchmark performance and training time was performed. It is important to note that in each episode of training 200 time-steps are taken. Discretising the state-action space into 10 angles, 10 velocities, and 10 torques will give a search space of 1000 cells. If we make a measure of how much samples are spent on searching in this search space, this sample count can be held constant with respect to increasing search spaces. This measure will treat larger search spaces with a proportionally longer training time to more fairly compare their outcomes. Setting the sample per state-action pair to e.g. 400, will set the number of episodes for a search space of 1000 to 2000 episodes (episodes x 200 timesteps = dθ x d$\dot{\theta}$ x dT x 400 samples → 2000 episodes). The relative experience per state-action pair is then kept constant between search spaces. Note that this is true on average, since not every state will have 400 visits because only visited states gain experience. With this measure, table 5.3 reviews the effect of increasing or decreasing the search space though action space reduction. Differences in benchmark scores are estimated from five runs of the training instance, observing random fluctuations in the outcome.

Table 5.3: Different training settings for the inverted pendulum with flat-RL. The number of samples per state-action pair (s/sap) is kept constant and thus determines the number of episodes used for training. Columns show discretisations for θ, $\dot{\theta}$, and the torque T.

Test case	# dθ	# d$\dot{\theta}$	# dT	Size search space	# s/sap	# Episodes	Benchmark score
1	25	25	15	9375	400	18750	-286 ± 18
2	25	25	10	6250	400	12500	-250 ± 12
3	25	25	6	3750	400	7500	-241 ± 38
4	25	25	4	2500	400	5000	-234 ± 12

Differentiating between exploration and exploitation is a delicate balance. For this various strategies exist. In section 3.5.2, ϵ-greedy, softmax and UCB were mentioned. Some research has been dedicated to coupling the greediness coefficient to the learning process, rather than empirically choosing one [59] [60]. Although the value-difference based exploration method [59] did show promising results in learning with more certainty, the consistency in outcome for this comparison wasn't satisfactory. For larger search spaces both methods are worth looking into, as well as other solutions such as sigmoid functions or other greediness curves. For the current test case the greediness was linearly increased from 0.2 to 0.9 throughout training (empirically determined). Encouraging exploration early on, but strengthening greediness later in the learning process gives a reliable learning curve. Greediness here is defined as a fraction where 1 = greedy, 0 = random.

The table shows that by decreasing the search space through action reduction, the performance of the algorithm is slightly improved with a lesser discrete action set. This effect can be because the average experience is kept constant, it is harder to find a general solution in a larger search space. Figure 5.4 shows that the solutions found are also roughly equal, but more discretisations gives more room for inaccuracy. The training time however, is largely affected. Spending 2500 episodes instead of 6250 for the same benchmark results in a decrease in training time of 60%. This time can be spent on further training the solution found in fewer episodes thereby enhancing its performance over the larger search space solution. Figure 5.4 shows a resulting time-traces of the two extreme solutions. The left figure has 25 discretisations, the right one four.

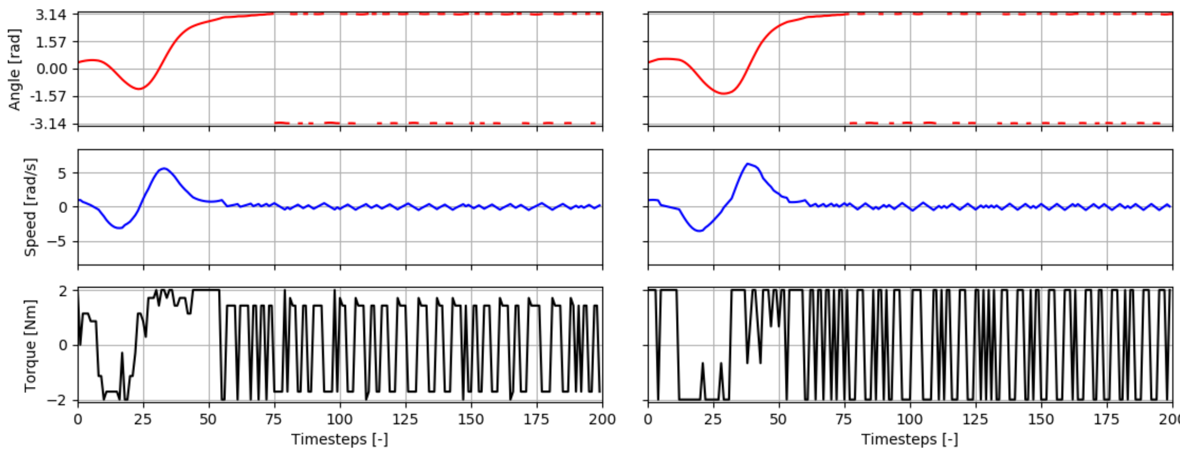


Figure 5.4: Flat-RL time traces showing angle, velocity, and torque curves over 200 time-steps of the simulated environment. Left figure shows the solution for 25 action discretisations. Right figure shows traces for four discrete actions only.

Note that in comparison to the PD controlled solution, flat-RL accomplishes a better benchmark result (-234 vs. -265). This is seen in both time-traces by the time taken to get velocity and angle in a steady, balanced state. For the PD controller this takes approximately 75 time-steps and with flat-RL about 60 steps. Since relatively little penalty is given for large torque inputs, flat-RL learns to give rather large and alternating torque inputs, even for 25 discretisations. This eventually results in an overall better benchmark score, proving the influence of the reward function on the outcome of a learning algorithm. When more penalty is added for application of torque, or changing thereof, the learning process is encouraged to give fewer inputs thereby imitating the PD controller more. Also the flat-RL result proves it is hard to manually design good control switch logic for a PD controller as in figure 5.2. Flat-RL proves more effective on the inverted pendulum task without much manual effort. Might the mass, drag or any other influential dynamic parameter of the pendulum change, the RL algorithm needs only to be run again and does not need complicated control theory for tuning. The discovered policy by the flat reinforcement learning algorithm can be displayed on a 2D matrix, showing the torques found for each angle-velocity combination that result in maximal return value. This is shown in figure 5.5:

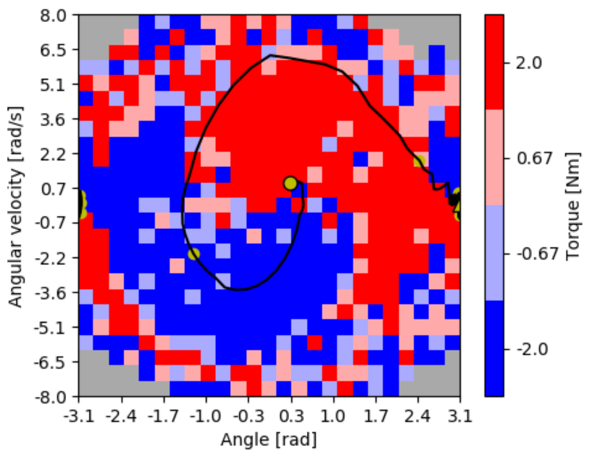


Figure 5.5: Flat-RL learned policy for the inverted pendulum swing-up task. A maximization over the 3D q-matrix, resulting in a 2D matrix showing the torques for each state giving maximal value return. The policy is overlayed with the traversed trajectory from initial to final condition where the middle represents the pendulum hanging down, an angle π away from balancing upright. Grey areas haven't been reached by the pendulum, no value information is available there.

Discretized angles and velocities are represented as square areas filled with the color of the associated torque found. A circle indicates the initial condition, each circle thereafter a 25-time-step increment, and the triangle finally the terminated state. As shown in the figure, the trajectory taken follows right along the edge of the red area, achieving minimum swing-up time. This edge marks the turn-over point of the applied torque, showing the delicate balance of the switching point. Also this policy hasn't completely converged still since multiple random spots are visible within the largely equally coloured areas. Even so, this policy is a good policy for the inverted pendulum task.

5.3. Temporal abstraction: HRL

An interesting remark that can be made about the time trace result of the torque curve of especially figure 5.3 but also the curves of figure 5.4 is the elongated application of torques over multiple timesteps. One could ask if it would be beneficial to include these temporally extended actions in the learning process by making them available as an option to choose from. Clearly, also in relation to the time-traces of the PD controller in figure 5.3, there are multiple occasions in which an action is applied over multiple time-steps. Especially in the swing-up phase the only useful action would be application of maximum absolute torque. Additionally, temporal extension allows for easier exploration of parts of the search space due to giving a prolonged action without minding the intermediate rewards. This is a method to overcome parts of the environment less easily crossed by a flat RL strategy.

Since sample efficient results have been obtained with the four action flat-RL method as shown in table 5.3, this section will consider hierarchical learning with four different actions. To show the effect of temporal abstraction on the learning process, a hierarchical training will be presented here where the algorithm can choose among the same four actions, but also choose for a temporal abstraction of those (extending multiple timesteps). To accomplish temporally extended courses of action, the four actions are formalised as four individual options, according to the options model by Sutton [20].

$$O = [-2, \quad -0.67, \quad 0.67, \quad 2] \tag{5.6}$$

Each option has an initiation set (I) equal to the entire state-space. Termination conditions are set likewise for each option. Choosing actions during the learning process is thereby reduced to picking actions on decision points. Setting good termination conditions is undeniably difficult. Many variants are possible. One idea is to make the decision points dependent on the reward function:

- Slope of reward signal changes sign → decide
- Slope of reward signal has reached highest point → decide
- Time in option > 10 time-steps → decide

Note that such conditions are not independent of the problem, since the reward signal is specific to the problem. This strategy might only work for problems where the reward signal is not sparse and always varying. With it, an undetermined amount of time-steps (up to 10) can be taken for each of the four actions. The learning algorithm thereafter decides which action is taken after termination of the previous. Figure 5.6 shows the time-trace of the solution. Blue filler indicates an action that is temporally extended, while white areas indicate single-step actions.

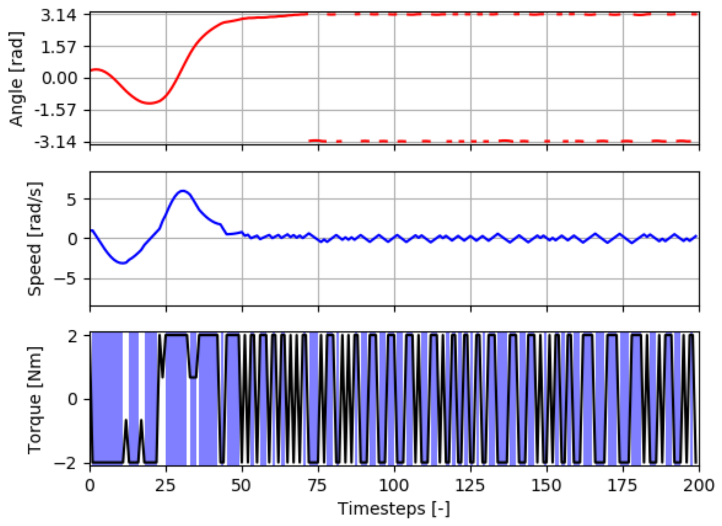


Figure 5.6: HRL time-trace plot showing angle, angular speed, and torque. Actions extending over a multitude of steps are coloured in blue, while single-step actions are shown in white.

As clearly visible, a large portion of the actions is now temporally extended. Most of this extension is forced by the termination conditions not triggering after a single step. Single step actions occur around decision points. The solution found is nearly identical to the time-traces of the PD-controller and the flat-RL solution. This HRL solution mostly shows the usefulness of temporal extension in the first part of the problem, where more easily a swing-up motion is found. After that, temporal extension and specifically these termination conditions become less useful to the solution. This is why the benchmark test results in a relatively low score of only -389 ± 73, significantly lower than both other methods. Key to effectiveness of HRL in a problem (with options) is therefore to design useful termination- and initiation conditions. Otherwise the problem may become harder to solve or might reach sub-optimal results. The policy associated with this time-trace is shown in figure 5.7, and shows nearly identical results with the flat-RL policy of figure 5.5. Note that execution of the torques directly from the shown policies is different for HRL, since the option logic is behind it, meaning there is an adherence to the initiation and termination conditions resulting in the temporally extended behaviour. Executing the hierarchical policy on a single-step basis as with flat-RL would alter the behviour of the solution found, making it incomparable to the outcome of training. Regardless, policies look very alike, with more solid coloured regions in the hierarchical solution since actions where executed for a prolonged time over these areas, resulting in a less fragmented result though this commitment.

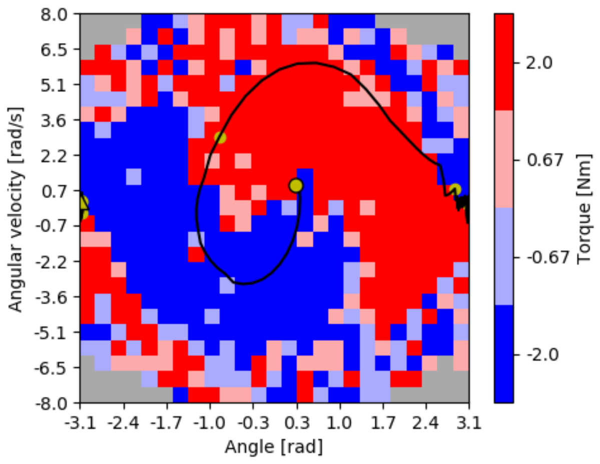


Figure 5.7: HRL policy plot for the inverted pendulum swing-up task. For each state the action resulting in the maximal value is shown. The underlying logic of options applies to this policy, meaning executing it is shown in a single-step manner would result in a different incomparable result. The policy is coupled to the way it was trained. The traversed trajectory from initial to final condition is overlayed where the middle represents the pendulum hanging down, an angle π away from balancing upright. Grey areas haven't been reached by the pendulum, no value information is available there.

5.4. Hybrid Hierarchical Reinforcement Learning

By including options in the learning process, also more complex extensions of actions can be thought of. Instead of only providing single- and multi-step options, they can also be defined as entire control structures by themselves. An obvious example of this would be the inclusion of a PD controller. This would effectively form a hybrid method, in which the classical control solution is combined with the HRL solution. A comparison to flat-RL in this case would be more difficult to make since the hybrid HRL solution is now capable of producing continuous actions, where the flat-RL approach is constrained to discretized actions only. The action space therefore is extended with respect to equation 5.6 with one option representing the PD controller:

$$O = [-2, \quad -0.67, \quad PD, \quad 0.67, \quad 2] \tag{5.7}$$

Using this hybrid method, a benchmark result of -238 ± 9 was achieved. This is very similar to the results obtained with flat-RL using the same amount of episodes for training. With one extra action used, thus complicating the problem as shown in table 5.3, this method proves more efficient than the flat approach. Figure 5.8 shows the time-traces of the pendulum executing on seed 0.

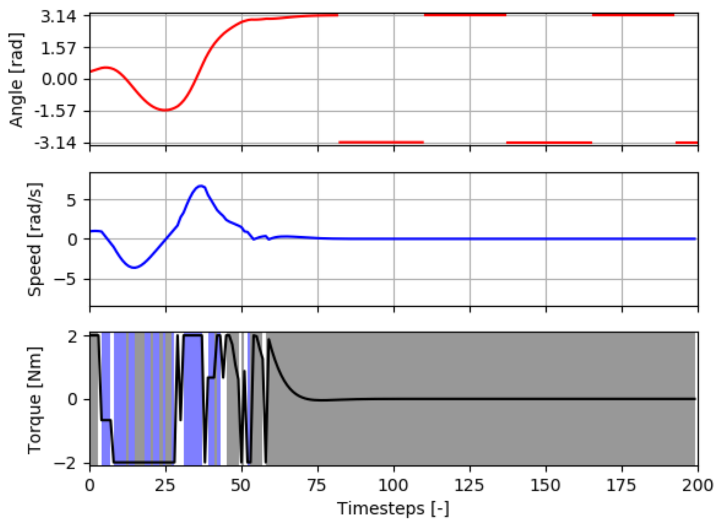


Figure 5.8: Hybrid HRL time-trace plot showing angle, angular speed, and torque. Actions extending over a multitude of steps are coloured in blue, single-step actions are shown in white, and PD control is shown in grey.

As shown, the temporal abstraction is a useful concept, especially in the swing-up phase. Afterwards, the inclusion of the PD controller is most useful. The hybrid hierarchical solution therefore combines the effectiveness of both methods, resulting in a better solution. Figure 5.9 shows the resulting policy of the hybrid solution. Two policy plots can be made. The first is the policy of actions, including the PD controller. To compare this policy to the other methods, the actions executed by the PD controller can be converted to the action space of size 4 used there. This is shown in the right plot of figure 5.9. It shows the again the similarities found, albeit with a different method.

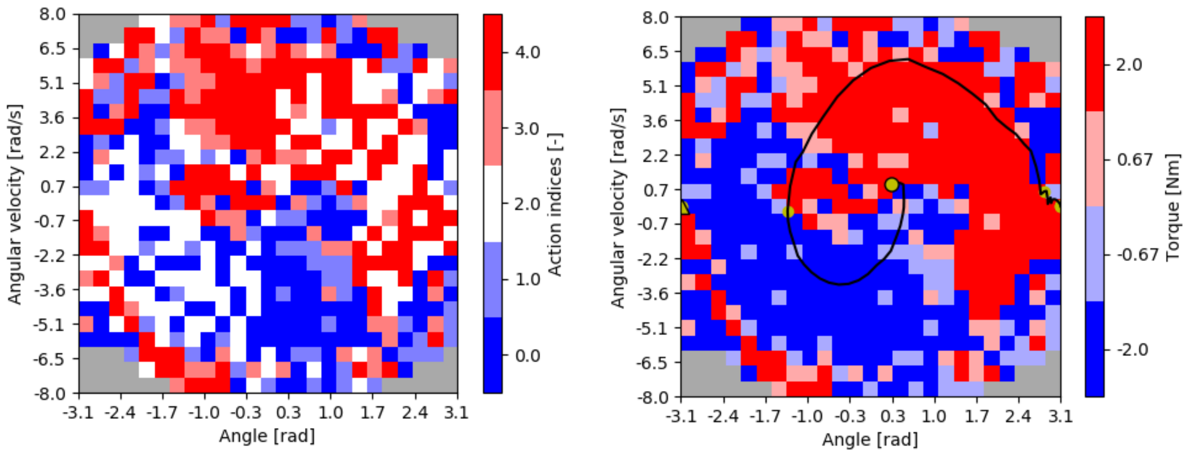


Figure 5.9: Hybrid HRL policy plot, showing the torques resulting in the maximum value. The left plot shows the policy including PD controller (white), and the right plot is a conversion to the discretized action space of size 4, similar to the flat-RL and HRL solutions.

5.5. Preliminary Analysis Results

This chapter was dedicated to the exploration of four different control methods for the problem of balancing an inverted pendulum. The four methods used here are: classical control, flat reinforcement learning, hierarchical reinforcement learning, and hybrid hierarchical reinforcement learning. A comparison between the discussed methods can be made. Table 5.4 shows the collected benchmark results:

Table 5.4: Comparison of benchmark test results for multiple control methods.

Method	Section	Action-space	# Episodes	Benchmark score
PD control	5.1	Continuous, swing-up and balance mode	N/A	-265 ± 0
Flat-RL	5.2	[-2 -.67 .67 2]	10000	-233 ± 13
HRL	5.3	[-2 -.67 .67 2] (temp. extended)	10000	-389 ± 73
Hybrid HRL	5.4	[-2 -.67 PD .67 2]	10000	-238 ± 9

Worst performance is found with the hierarchical method. The method was able to succesfully learn initially, but afterwards was limited by the termination conditions of its options. One can argue that sometimes it is easier to define control switch-points and PD controllers than it is to define good initiation- and termination conditions. The latter does not always have specific physical meaning and are therefore harder to design. In this case the tuning of these termination conditions meant no increase in performance past a certain level.

Slightly better performance is found with the classical PD control. Although its solution is successful on initialisations of the problem shown in this section, it fails to optimally reach the top balancing point on other occasions. This shows the difficulty of controller design manually, similar to the hand designed termination conditions in HRL. The finesse of the switch-over point and the controller itself are hard to obtain perfectly. Also, this manual design is valid only for this specific version of the pendulum. Might its length, mass, or friction change, the process needs to be repeated. Of course with more effort better performance can be obtained through this method, but the effort required is minimized through use of reinforcement learning.

Finally, roughly equal performance is achieved by both the flat reinforcement learning solution and the hybrid hierarchical reinforcement learning solution. Since the reward signal is non-sparse and differentiable, flat-RL has enough information to incrementally learn a solution. Might this not have been the case, it is possible that the search algorithm got stuck in certain parts where little information is available. For such problems HRL is a better alternative, since it can force the search though these areas more easily. As said before, the temporal extension of actions is useful in HRL especially in the beginning phases of the swing-up. Afterwards, the termination conditions hamper further increase in performance. Because the hybrid HRL method includes a PD controller for use after this phase, this reduction in performance is not seen. Even though the hybrid HRL method extends the action space by one over the flat-RL approach, an equal result in performance is eventually obtained. Counter-intuitive to the results shown in table 5.3. This demonstrates the effect of both reuse of known optimal control routines throughout the search, as well as the inclusion of temporal abstraction in certain situations.

Concluding, flat RL is a good solution to this problem since the reward function is non-sparse and differentiable. If either of these is not the case, HRL might provide a good solution to force the learning algorithm into other regions of the search space effectively. For this specific problem HRL was not able to outperform the flat-RL solution simply because the termination conditions of the algorithm are hard to set, and mostly limit the performance of the solution. This is in line with the findings in literature. However, the inclusion of a PD controller as an option to the learning algorithm, effectively creating a hybrid-hierarchical reinforcement learning algorithm was most effective. It showed that by using prior knowledge about effective controllers for parts of the search space the found solution outperforms other strategies.

The result obtained with this hybrid approach are promising for further development. The inclusion of optimal controllers in the action space does help the outcome of the learning process positively. Also, it is a natural idea to do this. If it is known that for a certain system a controller performs well in a certain region, but unclear is how control revolves around it, why would you not include this controller? Also, reinforcement learning is useful in this case to find the exact point of activation of this controller. The hierarchical component, introducing temporal abstraction, can aid in applying prolonged execution of activities, thereby helping traversing the search space. Knowing that the search space for an aircraft is much larger, and parts of it are hard to reach, the hybrid HRL approach is the best approach among the ones mentioned to take in the expansion towards the aircraft model.

6 Conclusions

The aim of this thesis is to answer the main research question:

"How can the sample efficiency of a reinforcement learning based flight controller for a fixed-wing aircraft be improved with hierarchical reinforcement learning by using a learning structure analogous to hierarchies in existing flight control methods?"

The order of the thesis report does not reflect the order of research. To maintain a structured answer to the main research question, conclusions are drawn per sub-research question:

1 How are flight control methods currently implemented?

- How is hierarchy present in classical flight control?

A study to the design of flight control reveals several types of hierarchies present throughout the aircraft. There are three levels of automatic control systems; the Stability Augmentation System (SAS), Control Augmentation System (CAS) and a variety of autopilots such as altitude hold mode, turn coordination and heading hold. All are layered according to their level of responsibility in an inner-outer loop structure and are part of the Automatic Flight Control System (AFCS). Other forms of hierarchy include state machines, gain scheduling and feedback control loops.

2 How can reinforcement learning be applied to flight control?

- What is state-of-the-art in flight control using reinforcement learning?
- What is state-of-the-art in flight control using hierarchical reinforcement learning?
- How can the hierarchical structure of existing flight control methods be converted into the framework of hierarchical reinforcement learning?
- How should a reward signal be shaped?

First a preliminary analysis on an inverted pendulum is conducted to familiarize with the application of Reinforcement Learning (RL) and specifically Hierarchical Reinforcement Learning (HRL) on a dynamical system. Four controller types are tested on the pendulum; a proportional-integral-derivative (PID) controller, HRL controller using *options*, and a hybrid *option* controller where one of the options was a proportional-derivative (PD) controller. From the initial experiment it is concluded that the effectiveness of *options* depends largely on the termination conditions. *Options* in the experiment are not most successful in balancing the pendulum, but the combination of *options* and PD-controller is. Flat-RL (FRL) also shows better performance than *options* alone.

Meanwhile a literature study was conducted on the topic of current applications of RL in flight control. The majority of of research is applied on Unmanned Aerial Vehicles (UAVs) where the common approach is to use offline learning first and thereafter improve performance online. Also these approaches derived either a model from expert demonstrations or from a model. Value-based methods are less popular. HRL for flight control is scarce, just two papers focussed on application of HRL to safe exploration. The reward signal used by the autonomous helicopter applications was the absolute deviation from the desired reference. Because of its lack of presence in literature this research contributes to the exploration of this method in this field.

To answer how an existing flight control structure can be implemented in HRL, the research was expanded to a simulated F16 aircraft. A linearised state-space model of the aircraft is implemented using Python to function as the environment for the agent. The task of altitude reference tracking is chosen and is split into three separate controllers, following a traditional altitude reference tracking controller. Four aircraft variables are of interest: the altitude h, flight path angle γ, pitch rate q and elevator deflection δ_e. The three controllers that link them in a logical structure are: $h-\gamma$, $\gamma-q$ and $q-\delta_e$. Learning is executed using Q-learning for both FRL, HRL and *options*. For *options* a different update rule is used according to the semi Markov Decision Process (SMDP) theory. A discretisation number of 25 is chosen for each parameter based on the minimum resolution this provides per variable.

3 What is the performance of the hierarchical reinforcement learning controller with and without temporal abstraction in relation to a controller based on flat reinforcement learning?

- What is the number of samples required to learn an altitude reference tracking controller using FRL and HRL?
- What is the tracking performance of the HRL controller in relation to FRL?
- What is the sample efficiency of the HRL controller in relation to flat reinforcement learning?

Three different control strategies are developed; FRL, HRL using a hierarchical decomposition according to a traditional three-layer altitude reference tracking controller, and HRL with the addition of *options*. FRL proved too inefficient to learn a control policy with the set number of 25 discretisations per state and action. After a reduction of the elevator deflection range from 12.4 degrees to 5, and many trails with different learning parameters, a policy is found using 119 million samples. Its validation tracking performance is -104.

HRL in comparison to FRL only required 9% of the samples whilst achieving an error 34% of that of FRL. The relative increase in sample efficiency is thereby of a factor 34. Using *options* further increases the sample efficiency since it requires even less samples, only 2%. The tracking performance of *options* however did not surpass HRL, achieving almost double the error. The overall relative sample efficiency of *options* is 81, showing the effectiveness of temporal abstraction. Similar to the conclusions of the preliminary analysis, *options* did not increase the performance of the controller. More research to the effect of termination conditions is required to enhance performance. Perhaps a different action selection method or learning parameter progression can aid this.

Although this answers all sub-questions, there are some additional findings. In addition to the decomposition and temporal abstraction a scaling of the discretisation domain is applied in an effort to further boost sample efficiency. The discretisation domain is linearly scaled towards the centre, effectively spreading experience samples over a broader range. It prevents imbalance during learning since state-visit dependent decays are used for the learning parameters ϵ and α, and increases the fineness of discretisation near the zero-error state. This can increase the performance in states where a successful agent is expected to spend most of its time without requiring more discretisations. More scaling results in increased skewing of visits towards other states. Too much scaling makes performance worse, but a little helps. A scaling of 0.9 was found most useful.

While the hierarchical decomposition made it possible to learn the altitude controller in the first place, it did triple the amount of hyper-parameters that are required to tune. Their initial value and decay are of big influence to the outcome of training and costly to tune when training can take several hours to complete. For each control layer a different reward delay and time-scale is applicable. This makes it difficult to set global learning parameters for each problem, when in fact they should be regarded as three separate problems.

Often the agent is found spending most samples on the edge of the discretisation domain. Because visit-dependent decay is used for α and ϵ, the agent would thereafter act greedy in those states and had difficulties escaping from that knowledge. Additional experiments with different action selection methods are required to find another solution to mitigate this problem.

This thesis showed the effectiveness of a hierarchical decomposition and the increase in sample efficiency. A further increase was observed when options where added. Future research is needed to continue in this direction, and could as a first step be directed towards expansion beyond longitudinal motion alone. Additionally, a static non-linear dynamic model was used but it is interesting to discover how the agent can be implemented in the presence of changing non-linear dynamics, which is possible with RL. More research is definitely needed to reduce some of the impact the learning parameters have on the outcome of training.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge, 1998. ISBN: 0 262 19398 1.

[2] Thomas G. Dietterich. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition". In: *Journal of Artificial Intelligence Research* 13 (2000), pp. 227–303. ISSN: 10769757. arXiv: `9905014 [cs]`.

[3] Richard Bellman. "A Markovian decision process". In: *Journal of Mathematics and Mechanics* (1957), pp. 679–684.

[4] Richard S. Sutton, Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial Intelligence* 112.1 (1999), pp. 181–211. ISSN: 00043702. arXiv: `arXiv:1011.1669v3`.

[5] Sridhar Mahadevan and Andrew G. Barto. "Recent Advances in Hierarchical Reinforcement Learning". In: *Event (London)* (2003), pp. 41–77. ISSN: 0924-6703.

[6] Karl Johan Åström and Tore Hägglund. *PID controllers: theory, design, and tuning*. Vol. 2. Instrument society of America Research Triangle Park, NC, 1995.

[7] Douglas J Leith and William E Leithead. "Survey of gain-scheduling analysis and design". In: *International journal of control* 73.11 (2000), pp. 1001–1025.

[8] Richard S. Russel. *Non-linear F-16 Simulation using Simulink and Matlab*. Version 1.0. 2003.

[9] Donald McLean. *Automatic flight control systems*. Vol. 16. Prentice Hall New York, 1990.

[10] Roger W Pratt. *Flight control systems*. American Institute of Aeronautics and Astronautics, 2000.

[11] Brian L Stevens, Frank L Lewis, and Eric N Johnson. *Aircraft control and simulation: dynamics, controls design, and autonomous systems*. John Wiley & Sons, 2015.

[12] E.H.J. Pallett and S. Coyle. *Automatic Flight Control*. Wiley, 1993. ISBN: 9780632034956.

[13] David Wyatt. *Aircraft Flight Instruments and Guidance Systems: Principles, Operations and Maintenance*. Routledge, 2014.

[14] Mohammad Sadeghi, Alireza Abaspour, and Seyed Hosein Sadati. "A novel integrated guidance and control system design in formation flight". In: *Journal of Aerospace Technology and Management* 7.4 (2015), pp. 432–442.

[15] Luat T Nguyen et al. "Simulator study of stall/post-stall characteristics of a fighter airplane with relaxed longitudinal static stability.[F-16]". In: (1979).

[16] Brian L. Stevens and Frank L. Lewis. *Aircraft Control and Simulation*. Vol. 76. Jan. 1992.

[17] Dale Enns et al. "Dynamic inversion: an evolving methodology for flight control design". In: *International Journal of control* 59.1 (1994), pp. 71–91.

[18] S Sieberling, QP Chu, and JA Mulder. "Robust flight control using incremental nonlinear dynamic inversion and angular acceleration prediction". In: *Journal of guidance, control, and dynamics* 33.6 (2010), pp. 1732–1742.

[19] Karl J Åström and Björn Wittenmark. *Adaptive control*. Courier Corporation, 2013.

[20] Richard S Sutton, Andrew G Barto, and Ronald J Williams. "Reinforcement learning is direct adaptive optimal control". In: *IEEE Control Systems* 12.2 (1992), pp. 19–22.

[21] Robert F Stengel. "Toward intelligent flight control". In: *IEEE transactions on Systems, Man, and Cybernetics* 23.6 (1993), pp. 1699–1717.

[22] Christopher John Cornish Hellaby Watkins. "Learning from delayed rewards". PhD thesis. King's College, Cambridge, 1989.

[23] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[24] Robert H Crites and Andrew G Barto. "Improving elevator performance using reinforcement learning". In: *Advances in neural information processing systems*. 1996, pp. 1017–1023.

[25] Ronald Edward Parr. "Hierarchical Control and Learning for Markov Decision Processes". In: (1998).

[26] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[27] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[28] David Silver et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[29] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[30] Richard Bellman. *The theory of dynamic programming*. Tech. rep. RAND Corp Santa Monica CA, 1954.

[31] Lucian Busoniu et al. *Reinforcement learning and dynamic programming using function approximators*. CRC press, 2010.

[32] Kai Arulkumaran et al. "A brief survey of deep reinforcement learning". In: *arXiv preprint arXiv:1708.05866* (2017).

[33] Vijay R Konda and John N Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[34] Silver D. *Introduction to Reinforcement Learning*. 2015.

[35] Ivo Grondman et al. "A survey of actor-critic reinforcement learning: Standard and natural policy gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307.

[36] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[37] Marco Wiering and Martijn Van Otterlo. "Reinforcement learning". In: *Adaptation, learning, and optimization* 12 (2012), p. 51.

[38] Kai Arulkumaran et al. "A Brief Survey of Deep Reinforcement Learning". In: (2017), pp. 1–16. ISSN: 1701.07274. DOI: `10.1109/MSP.2017.2743240`. arXiv: `1708.05866`. URL: `http://arxiv.org/abs/1708.05866{\%}0Ahttp://dx.doi.org/10.1109/MSP.2017.2743240`.

[39] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[40] Silvia Ferrari and Robert F Stengel. "Online adaptive critic flight control". In: *Journal of Guidance, Control, and Dynamics* 27.5 (2004), pp. 777–786.

[41] Andrew Y Ng et al. "Autonomous inverted helicopter flight via reinforcement learning". In: *Experimental Robotics IX*. Springer, 2006, pp. 363–372.

[42] Jaime Junell et al. "Self-tuning Gains of a Quadrotor using a Simple Model for Policy Gradient Reinforcement Learning". In: *AIAA Guidance, Navigation, and Control Conference*. 2016, p. 1387.

[43] Malcolm Ross Kinsella Ryan. *Hierarchical reinforcement learning: a hybrid approach*. Citeseer, 2002.

[44] Debbie Yee et al. "Optimal Behavioral Hierarchy". In: 10.8 (2014).

[45] Raja Giryes and Michael Elad. "Reinforcement Learning: A Survey". In: *Eur. Signal Process. Conf.* (2011), pp. 1475 –1479. ISSN: 10769757. arXiv: `9605103 [cs]`.

[46] Steven J Bradtke and Michael O Duff. "Reinforcement learning methods for continuous-time Markov decision problems". In: *Advances in neural information processing systems*. 1995, pp. 393–400.

[47] Andrew G. Barto and Sridhar Mahadevan. "Recent Advances in Hierarchical Reinforcement Learning". In: *Event (London)* (2003), pp. 41–77. ISSN: 0924-6703.

[48] Peter Dayan and Geoffrey E Hinton. "Feudal reinforcement learning". In: *Advances in neural information processing systems*. 1993, pp. 271–278.

[49] Nuttapong Chentanez, Andrew G Barto, and Satinder P Singh. "Intrinsically motivated reinforcement learning". In: *Advances in neural information processing systems*. 2005, pp. 1281–1288.

[50] Richard S Sutton, Doina Precup, and Satinder P Singh. "Intra-Option Learning about Temporally Abstract Actions." In: *ICML*. Vol. 98. 1998, pp. 556–564.

[51] Tejas D Kulkarni et al. "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation". In: *Advances in neural information processing systems*. 2016, pp. 3675–3683.

[52] Pierre-Luc Bacon, Jean Harb, and Doina Precup. "The Option-Critic Architecture." In: *AAAI*. 2017, pp. 1726–1734.

[53] Alexander Sasha Vezhnevets et al. "Feudal networks for hierarchical reinforcement learning". In: *arXiv preprint arXiv:1703.01161* (2017).

[54] Chunlin Chen, Han-Xiong Li, and Daoyi Dong. "Hybrid control for robot navigation-a hierarchical Q-learning algorithm". In: *IEEE Robotics & Automation Magazine* 15.2 (2008).

[55] Andrew Levy, Robert Platt, and Kate Saenko. "Hierarchical Reinforcement Learning with Hindsight". In: *arXiv preprint arXiv:1805.08180* (2018).

[56] Stephen Verbist, Tommaso Mannucci, and Erik-Jan Van Kampen. "The Actor-Judge Method: safe state exploration for Hierarchical Reinforcement Learning Controllers". In: *2018 AIAA Information Systems-AIAA Infotech@ Aerospace*. 2018, p. 1634.

[57] Oliver Kroemer et al. "Towards learning hierarchical skills for multi-phase manipulation tasks". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, pp. 1503–1510.

[58] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[59] Michel Tokic. "Adaptive epsilon-Greedy Exploration in Reinforcement Learning Based on Value Differences". In: Sept. 2010, pp. 203–210.

[60] Alexandre dos Santos Mignon and Ricardo Luis de Azevedo da Rocha. "An Adaptive Implementation of epsilon-Greedy in Reinforcement Learning". In: *Procedia Computer Science* 109 (2017). 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal, pp. 1146 –1151.