Intelligent Flapping Wing Control

Reinforcement Learning for the DelFly

M.W. Goedhart

Technische Universiteit Delft

TU Delft

Intelligent Flapping Wing Control

Reinforcement Learning for the DelFly

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

M.W. Goedhart

June 7, 2017

Faculty of Aerospace Engineering · Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

Dated: June 7, 2017

Readers:

Dr. ir. E. van Kampen

Dr. ir. C. C. de Visser

Dipl.-Ing. S. F. Armanini

Dr. O. A. Sharpanskykh

Dr. ir. Q. P. Chu

Preface

This reports concludes the work conducted for my thesis. In the graduation phase at the department of Control and Simulation at the faculty of Aerospace Engineering of Delft University of Technology, a phase of academic research is formalized in the course AE5310 Thesis Control and Operations.

This report is intended to show the knowledge gained during the thesis phase. The main scientific contributions of this project are described in a scientific paper, which is included in this thesis. This is intended as a conference paper. It is aimed at readers with a background in intelligent aerospace control, and is therefore written at a more difficult level. It should be readable as a standalone document.

For readers without previous knowledge of Reinforcement Learning control, a literature survey is summarized in this thesis. This should be sufficient to understand the paper. Such readers are advised to read Chapter 6 to 9 before continuing to the paper.

I would like to thank dr. Erik-Jan van Kampen, dr. Coen de Visser and Sophie Armanini, my supervisors on this thesis project. This work would not have been possible without their support and valuable feedback. Also, I would like to thank my family for supporting me during the months I spend on my thesis, and the years of study before that.

M.W. Goedhart
Zwijndrecht, May 2017

Contents

1 Introduction

The DelFly [11] is a Micro Air Vehicle (MAV) that generates lift by flapping its wings. It has been capable of flight since 2005. Several authors have been able to develop automatic flight control systems for the DelFly by means of PID control [10–12, 23, 48]. However, these controllers are all limited to specific flight conditions. Only for flight in a wind tunnel, more advanced control concepts have been applied [7, 8].

Due to the small size of the DelFly, manufacturing is very complicated, which causes variations between individual vehicles [21]. In practice, it is found that the gains of the controllers need to be hand-tuned for each individual DelFly, which is time-consuming [21].

The application of more advanced controllers has been limited by the unavailability of accurate aerodynamic models. The first linear model was published only in 2013 [5]. Research efforts have led to accurate models for specific flight conditions [2], but the identification of a global model until recently has been challenging. Classical control uses frequency or Laplace domain techniques to develop controllers for linear systems. In modern control, time domain representations are used. This has made it possible to handle nonlinear systems as well. However, these controllers are fully predefined by the control engineers who designed them. Controllers that are able to adapt their behavior could be seen as a new phase in control engineering [26]. Adaptive controllers contain a parametrized representation of their behavior. Algorithms are used to change the parameters during interaction with the system.

Reinforcement Learning (RL) control tries to mimic the learning methods that humans and animals use [46]. Reinforcement Learning is a machine learning technique where the intelligent system learns by trial and error. Historically, RL has been used to study the learning behavior of animals [43]. Reinforcement Learning is inherently based on optimal control [43]. The intelligent agent is always trying to optimize some function. From a control engineer's perspective, it provides a trial and error approach to optimal control, where the performance of a poor controller is improved continuously. The strength of Reinforcement Learning methods is that they can often deal with an inaccurate model of the controlled system. This makes RL suitable for problems where accurate models are not available [21].

The challenges of limited scope of the models and variability of the DelFly may be effectively tackled by Reinforcement Learning [21]. Reinforcement Learning has been applied to flapping wings to maximize lift [33, 34, 41, 50], but has never been used for flight control of Flapping Wing MAVs. In this graduation project, a Reinforcement Learning controller will be developed in order to achieve automatic flight of the DelFly.

A detailed explanation of the scope, challenges and phases of this project is presented in Chapter 2. The main contributions to science are described in a scientific paper, contained in Chapter 3. Chapter 4 provides some additional results and reflection on the methods and results of the paper. Some of the recommendations from Chapter 3 are analyzed further in Chapter 5.

After the results have been presented, the road towards them will be explained. Chapter 6 provides an introduction to the field of Reinforcement Learning. A review of Reinforcement Learning literature for flight control applications is performed in Chapter 7. Some important implementation considerations are discussed in Chapter 8. Chapter 9 describes how the preselection of algorithms from literature is narrowed down to the algorithms considered in the paper. This chapter also presents some knowledge that is obtained during the preliminary tests. The scientific conclusions and recommendations are contained in the paper in Chapter 3, but the project itself is wrapped up in Chapter 10.

2

Research scope and framework

How does one control a Flapping Wing Micro Air Vehicle (FWMAV) with intelligent control techniques? This question nicely captures the aim of this project, but is much too broad to be a suitable research question. This chapter will discuss the exact content of this thesis project. Section 2.1 explains what aspects will be investigated. A set of research questions is synthesized from this in Section 2.2.

2.1. Scope of the project

The DelFly is a Flapping Wing MAV. It has been designed using a top-down approach [10], meaning that a functioning larger vehicle was miniaturized. As a result, the complex dynamics of the DelFly are still not fully understood. It is known that the behavior is nonlinear, and the flapping motion induces time-varying components. A successful Reinforcement Learning (RL) controller must be able to handle this complex dynamic system.

The small size of the DelFly makes it difficult to manufacture. This results in large inconsistencies between different individual DelFly's. This is the reason why intelligent control is suggested as a feasible option. However, the variability between different vehicles still poses a large challenge to the controller, which is solved in this project.

The DelFly has been used as a platform for artificial intelligence research [11]. Its focus on autonomous control is reflected by the availability of a camera. Vision-based control algorithms are commonly employed for the DelFly (e.g., [44]), leading to fully autonomous flight outside the laboratory.

In this project, the focus is on automatic control, rather than autonomous control. It was decided that vision-based control should not be the focus of this project. This is because it is mostly concerned with extracting information from camera data, rather than with flight control. Instead, external tracking equipment is used to determine the state of the DelFly. This is done by using feedback from the Delft University of Technology Cyber Zoo, an optical tracking facility which uses infrared cameras to track the position of objects. Additionally, data from the Inertial Measurement Unit on the DelFly is used. These measurements can be considered more reliable than vision-based measurements, but are still subject to significant noise. Dealing with this noise is one of the challenges of this project.

Using feedback from the Cyber Zoo requires external computer systems to be connected to the DelFly in real time. This opens the possibility of computing the control off-board as well. This would alleviate the challenge of the computational loads of RL algorithms. On-board control is considered preferable, but optional in this research. Nevertheless, the ultimate goal of on-board control is considered in the judgment of the RL algorithms. Computational time is not formally assessed, but rated qualitatively as feasible or infeasible.

This project is about Reinforcement Learning as a control method. Research efforts are focused on RL control. Other adaptive control topologies are not considered in this study. Learning from a human example in a supervised setting is not the intention.

In an automatic control setting, the controller regulates the state of the vehicle to a certain reference. This reference could be set by a human pilot, such that the controller functions as a control augmentation system. However, manual handling qualities are not considered in this thesis. Rather, the trajectory is a predefined step.

2.2. Research questions

The challenges of automatic flight control of the DelFly are its complex, time-varying dynamics, variability in its dynamics, noisy data and the limited accuracy of aerodynamic models. In this project, Reinforcement Learning is investigated as a control method to tackle these problems. Since the main uncertainty is in the performance of such a controller, the main research question is defined as follows:

What is the performance of a Reinforcement Learning controller for the DelFly, trained using dynamic models, compared to conventional controllers?

This research question highlights the main characteristic of the project, which is the fact that a RL controller will be trained on aerodynamic models. In order to answer the main question, several sub-questions will have to be answered:

- Which measures are suitable to compare the performance of the controllers?
- What is the performance of Proportional-Integral-Derivative (PID) control of the DelFly?
- What is the best method of implementing Reinforcement Learning control for the DelFly?
  - What Reinforcement Learning controllers are available for flight control applications?
  - How do high-level indirect controllers compare to low-level direct controllers in terms of safety and accuracy?
  - Which representation of the flight state in the controller leads to the best performance in terms of learning speed, accuracy and safety?
  - Which Reinforcement Learning methods are computationally too expensive for practical use?
- What is the best method of using the existing dynamic models of the DelFly to train the controller of the actual DelFly?
  - Which dynamic models of the DelFly are available?
  - What is the best way of training a global controller on several local models?
  - What is the best method to switch from offline to online learning?

When these questions are answered, the answer to the main research question should be known. The lower level questions are intended to provide a steering function, whereas the questions on a higher level focus more on reaching the research objective. Note that the term 'performance' in the main research question is currently not well-defined. The first question focuses on finding a performance metric that includes both safety and accuracy. The third question, with its associated sub-questions, attempts to gather a set of possibly feasible algorithms and narrow it down to a small selection. Of course, many algorithms are available, and not every subtle difference should be reflected by the research questions. The trade-off between direct and indirect control should be addressed in any case. Also, it is important to consider the way in which the the flight state is represented in the controller. It is possible to use the tracking error, which is common in linear controllers, or the full state. Alternatively, one may use certain features of the state, such as the distance from the walls instead of the vehicle's position, to facilitate processing. This is especially interesting for low-level controllers. For controllers on a higher level, this research question relates to the structure of the underlying low-level controller. A detailed analysis of computational requirements is not part of this project, but algorithms that are clearly too expensive for practical use are disregarded.

The last research questions are indicated in gray. They were originally intended to steer the research into low-level controllers. A preliminary analysis into this area, described in Chapter 9, was inconclusive. Because low-level algorithms were later disregarded, answering these questions was no longer necessary in order to find the answer to the main question.

Note that a flight test is not strictly necessary in order to answer the research question. However, a flight test would increase the confidence in the results, and is therefore considered desirable. The research objective is formalized below:

The research objective is to design and test a Reinforcement Learning controller for the DelFly, and to compare it to existing controllers, such that recommendations can be made.

This project is design-oriented, but also serves an evaluative function by making general conclusions on RL control for this application platform. The plan is to perform a rigorous application study, which will highlight solutions to the current problems, but will also highlight new problems. This will be done mostly by applying recent RL solutions. The following sub-goals can be distinguished:

- Make an overview of the current state-of-the-art in Reinforcement Learning control.
- Make a selection of control methods that seem feasible for the DelFly.
- Refine this selection by means of preliminary research.
- Design a set of Reinforcement Learning controllers for the DelFly.
- Determine which existing controllers are to be used as a baseline for the performance.
- Compare the performance of the selected controllers in simulation.
- Compare the performance of a smaller selection of controllers in flight tests.
- Derive recommendations from an analysis of the performance.

The sub-goals show that multiple steps are taken to find the best RL controller. From a literature survey, a set of possible controllers is selected for detailed design. After an analysis in simulation, a new selection is be made for the final implementation. The flight test has not been performed during the course of this project, as will be described in Section 2.4.

When the research objective is achieved, the result will be the world's first Reinforcement Learning flight controller for a Flapping Wing MAV, with recommendations for performance improvements. Also, a new automatic controller for the DelFly will be available. This will facilitate future research on autonomous control and Flapping Wing aerodynamics.

2.3. Challenges of DelFly control

In this project, the goal is to design a controller for the DelFly. This section will consider the design problem from the controller's point of view.

The system to be controlled is a Flapping Wing MAV. It has three independent control inputs: the elevator, rudder and flapping frequency. It was established in Chapter 2 that it will be controlled within a special tracking environment. This removes the need for vision-based control algorithms, and provides the possibility of off-board control, alleviating the challenges of computational requirements. Four important challenges are distinguished:

1. Variability in dynamics
   Inconsistencies in manufacturing make every DelFly unique. This makes it difficult to design a controller that works for every DelFly. The first solution to this challenge is an *adaptive* controller. Such a controller is optimized for one DelFly, and adapts its behavior when it is installed on a new individual. The different DelFly's should have similar dynamics for this to work. An alternative approach is to design a fixed, but *robust* controller. This single controller is robust enough to deal with variations from the nominal DelFly. In this report, the robust category is defined as a controller that makes use of pre-defined control solutions. The resulting behavior does not have to be constant: it may switch between several solutions. The characteristic of the robust category is that the pre-defined solutions are fixed: selection occurs on a higher control level.

2. Complex dynamics
   The dynamics of the DelFly are complex in two aspects. When considering the entire flight envelope, the **nonlinearity** of the DelFly complicates control. This can be countered with complex controller structures, but it is preferred to use simple structures if those allow good performance. The challenge in this project is to design a controller of suitable complexity. It should be noted that the selected task will influence the required complexity. This is because some tasks, like aggressive maneuvers, may involve a large part of the flight envelope, while simple tasks only consider slight deviations from the trimmed condition. Another point to note is that the **flapping** behavior causes additional motion. This motion could be considered a disturbance, which is dealt with like any other disturbance from the

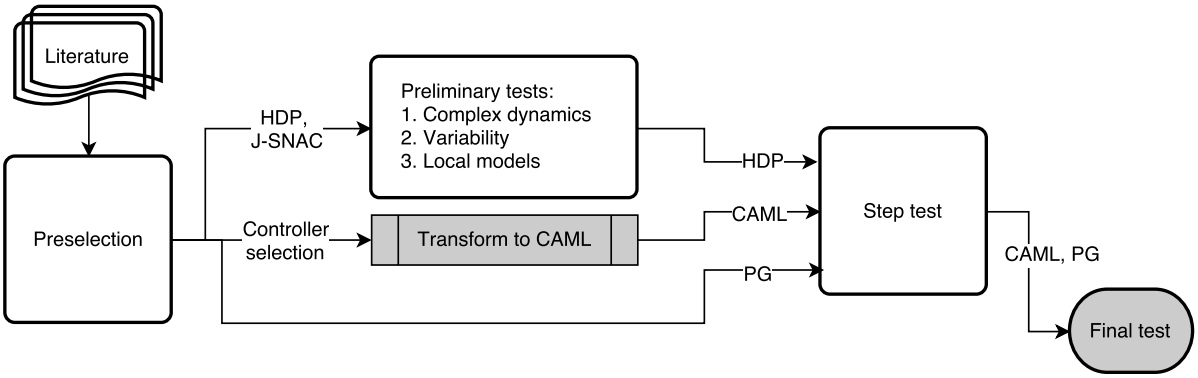


Figure 2.1: Flowchart describing the phases of the project. Gray boxes are elaborated in the paper.

required flight state. However, it may be necessary to actively counteract the time-varying motion. It is also possible to disregard the flapping motion, by controlling only the 'time-averaged' state. This can be done by filtering the perceived state of the DelFly. Research is required to ensure that such filters do not influence the time-averaged behavior of the DelFly too much.

3. Noise
   Although the optical tracking facility removes the need for vision-based control, it is not perfect. The measurements by the Inertial Measurement Unit (IMU) and the TU Delft Cyber Zoo are subject to noise. The controller should be able to deal with noisy measurements.

4. Disturbances
   The TU Delft Cyber Zoo is a laboratory environment. However, it is not a wind-free environment. The controllers must be able to deal with some external disturbances to the state.

The tools available to tackle these challenges are the **local models** of the dynamics of the DelFly. Several Linear Time-Invariant models are known, which are valid at specific points of the flight envelope. However, a global model of the dynamics is not known. Using the local models to obtain a global controller is a challenge of this project.

2.4. Project procedure

The previous sections listed the goals, challenges and intended steps of this project. As the project progressed, it was found that some steps were unnecessary, and other items were missing. This section describes the steps that have been followed in the project.

A flight test in the Cyber Zoo was originally intended as the final part of this project. Near the intended start of this phase, however, it was decided that continuing the simulated tests would contribute more to the body of science than a flight test. A flight test is left as the main recommendation for validation of this research. This thesis considers only a simulation of such a flight test.

This project was intended to compare the performance of Reinforcement Learning controllers to the performance of PID controllers. Developing new control approaches was considered optional. During the course of the project, the Classification Approach for Machine Learning control (CAML) was developed. This is a control approach relying on classification networks. Traditionally, Machine Learning is subdivided into Supervised Learning, Unsupervised Learning, and Reinforcement Learning. CAML does not fit well in any of these categories. A classifier is trained in a supervised setting, using simulated experience. This is then applied in combination with a model that is learned online. CAML is therefore a hybrid form of Supervised and Reinforcement Learning.

The main steps of the project are shown in Figure 2.1. The project starts with a literature study. A preselection of promising RL algorithms is made from literature. Next, the algorithms are subjected to some preliminary tests, in order to make a final selection. At this stage, the CAML approach was also initiated. The Policy Gradient (PG) algorithm was not suitable for the preliminary tests, but good performance was found in literature, and during informal analyses. It was therefore used during the next tests.

The *Step* test requires the controllers to track a velocity step by deflecting the elevator. This is not very different from the final test; it is in fact an early version. One more algorithm was discarded during this test,

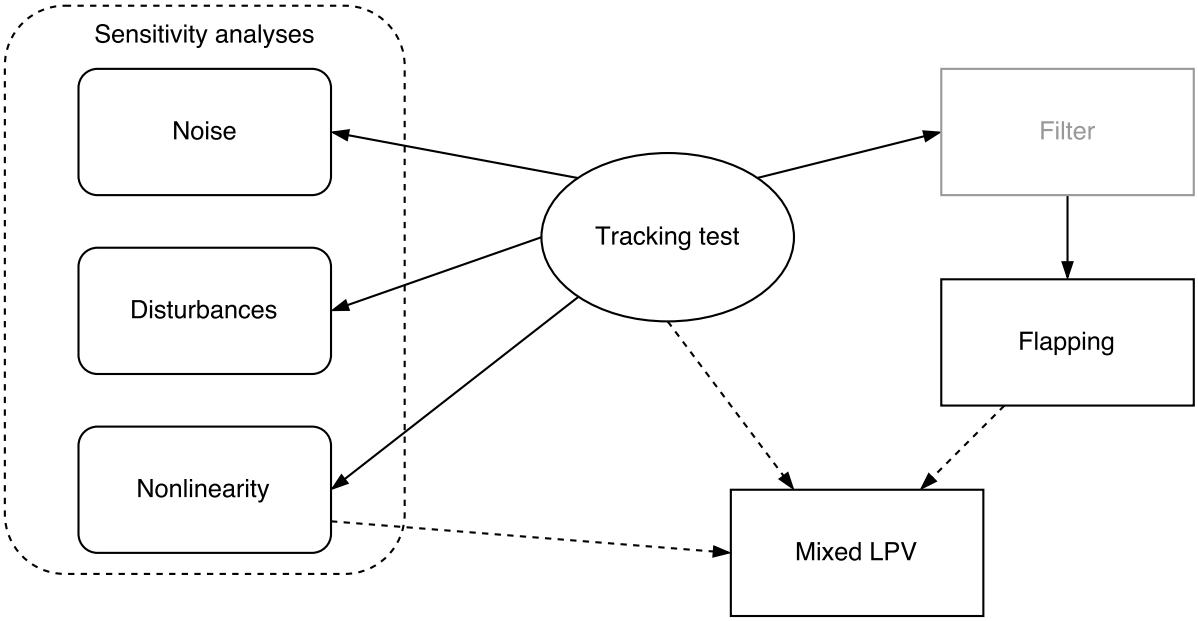


Figure 2.2: Simulation steps of the final test.

and only two algorithms were used for the final test.

The steps of the final test are shown in Figure 2.2. The central *Tracking* is also a test for velocity tracking performance, but additionally punishes poor damping. This test is used as the baseline for a number of other analyses. Three sensitivity analyses are performed, to assess the sensitivity of the algorithms to noise, disturbances and nonlinearities. These tests only consider the time-averaged part of the model. It is later assessed how well the algorithms perform if flapping is added. A low-pass filter is required for effective performance of the controllers. An intermediate step is to add only this filter, without including the flapping motion itself.

The last analysis of this project is the *Mixed LPV* test. This test assesses the performance when flapping, nonlinearity and noise are combined. This test is performed on one global model, which has a Linear Parameter-Varying structure.

The steps of the project are reflected in the structure of this report. The scientific content starts with the most important contributions to the body of science. These are described in a paper, intended for presentation at a scientific conference. This paper is included in Chapter 3. The paper describes the gray blocks in Figure 2.2. The results of the final test and the development of the CAML algorithm are considered important. Some additional results are described in Chapter 4. The road towards these results is described afterwards.

3

Scientific paper

This chapter contains the scientific paper that was written as part of the thesis process. This paper should be readable as a standalone document. It is intended to be presented at a scientific conference.

Machine Learning for Flapping Wing Flight Control

Menno W. Goedhart*, Erik-Jan van Kampen†, Sophie F. Armanini‡,
Coen C. de Visser† and Qiping Chu§

Delft University of Technology, 2629 HS Delft, The Netherlands

Flight control of Flapping Wing Micro Air Vehicles is challenging, because of their complex dynamics and variability due to manufacturing inconsistencies. Machine Learning algorithms can be used to tackle these challenges. A Policy Gradient algorithm is used to tune the gains of a Proportional-Integral controller using Reinforcement Learning. A novel Classification Algorithm for Machine Learning control (CAML) is presented, which uses model identification and a neural network classifier to select from several predefined gain sets. The algorithms show comparable performance when considering variability only, but the Policy Gradient algorithm is more robust to noise, disturbances, nonlinearities and flapping motion. CAML seems to be promising for problems where no single gain set is available to stabilize the entire set of variable systems.

Nomenclature

c	Vertical speed, m/s
F	Force, N
f	Flapping frequency, Hz
g	Gravitational acceleration, m/s^2
I	Moment of inertia, kg m^2
J	Cumulative reward, -
m	Mass, kg
q	Pitch rate, rad/s
S	Score, -
U	Reward, -
u	Control input, -
$\mathfrak{u}$	Out-of-plane velocity, m/s
V	Horizontal speed, m/s
w	Axial velocity, m/s
x	State
α	Angle of attack, rad
δ	Deflection, rad
θ	Pitch angle, rad
ϕ	Nonlinearity, -

Subscript

d	Disturbance
e	Elevator
n	Noise

*MSc Student, Control & Simulation Section, Faculty of Aerospace Engineering, Kluyverweg 1, Delft, The Netherlands.

†Assistant Professor, Control & Simulation Section, Faculty of Aerospace Engineering, Kluyverweg 1, Delft, The Netherlands. AIAA Member.

‡PhD Student, Control & Simulation Section, Faculty of Aerospace Engineering, Kluyverweg 1, Delft, The Netherlands. AIAA Student Member.

§Associate Professor, Control & Simulation Section, Faculty of Aerospace Engineering, Kluyverweg 1, Delft, The Netherlands. AIAA Member.

I. Introduction

The DelFly[1] is a Micro Air Vehicle (MAV) that generates lift by flapping its wings. It has been capable of flight since 2005. Several authors have been able to develop automatic flight control systems for the DelFly by means of PID control.[1–4] However, these controllers are all limited to specific flight conditions. Only for flight in a wind tunnel, more advanced control concepts have been applied.[5]

Due to the small size of the DelFly, manufacturing is very complicated, which causes variations between individual vehicles.[6] In practice, it is found that the gains of the controllers need to be hand-tuned for each individual DelFly, which is time-consuming.[6]

The application of more advanced controllers has been limited by the unavailability of accurate aerodynamic models: the first linear model was published only in 2013.[7] Research efforts have led to accurate models for specific flight conditions,[8] but the identification of a global model until recently has been challenging.[9] Adaptive or robust controllers are required to deal with the variability and nonlinearity of the DelFly.

Reinforcement Learning (RL) control tries to mimic the learning methods that humans and animals use.[10] Reinforcement Learning is a Machine Learning (ML) technique where the intelligent system learns by trial and error. Historically, RL has been used to study the learning behavior of animals.[11] From a control engineer's perspective, it provides a trial and error approach to optimal control, where the performance of a poor controller is improved continuously. The strength of Reinforcement Learning methods is that they can often deal with an inaccurate model of the controlled system. This makes RL suitable for problems where accurate models are not available.[6] Reinforcement Learning has been applied to flapping wings to maximize lift,[12–15] but has never been used for flight control of Flapping Wing MAVs.

This paper is a step towards a more intelligent controller for the DelFly, using Machine Learning. RL control theory is employed to develop a Proportional-Integral (PI) controller that automatically tunes its gains after gathering flight experience.[6,16] Additionally, the Classification Algorithm for Machine Learning control (CAML) is proposed, a novel classification controller where a Neural Network (NN) is used to select the gains from a predefined set. This is compared to the performance of a conventional PI controller with fixed gains. Simulations are performed to assess the performance of these algorithms.

The following structure is used in this paper. Section II describes the problem of controlling the DelFly, which is used to compare the algorithms. The Policy Gradient (PG) algorithm is described in section III, and CAML in section IV. The results of the comparison are presented in section V. Section VI concludes this work. The appendix describes how the nonlinearity measure used in this research is defined.

II. Problem Description

This research is both a step towards a new controller for the DelFly, and a comparison of Machine Learning control algorithms. For both goals, it is essential to understand the control problem, which is described in this section. The system to be controlled, on which the models used in this research are based, is shown in figure 1.

Several challenges were identified for the control of the DelFly. First of all, the *variability* between different individual vehicles calls for adaptive or robust control methods. Second, the dynamics of the DelFly are *nonlinear*. Third, the *flapping* motion of the wings influences control. Finally, *noise* and *disturbances* affect the possible controller performance.

This research focuses on automatic control: the vision-based control architecture is not used. As an intermediate step[a], a simulation of flight with feedback from an optical tracking system is performed. This research only considers control of the elevator. The throttle and rudder are not used in the simulations, because accurate models including these actuators are not available.

II.A. Dynamic Models of the DelFly

The dynamic models of the DelFly are the tools available to solve the challenges. Currently, these are mostly *local models*, which are valid in only one flight condition. The model structure[8] considered in this paper splits the dynamics into a time-averaged part, which is a Linear Time-Invariant (LTI) model, and a time-varying

[a]The DelFly is equipped with a drift-sensitive Inertial Measurement Unit, and the optical tracking system provides drift-free state measurements. Such measurements could be provided outside the laboratory by the on-board camera or satellite navigation, but this is more complex.

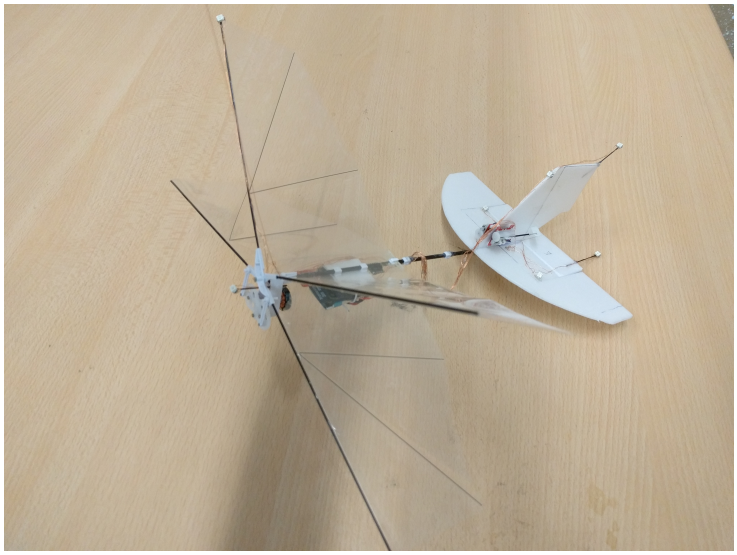


Figure 1: Variant of the DelFly II, on which the simulations in this research are based.

part, which is a Fourier series approximating the forces due to the repetitive flapping motion. These parts are used to match low-pass and high-pass filtered versions of flight data, respectively. The structure of the time-averaged model part is shown in Eq. 1.[17] The bias parameters b_1, b_2, and b_3 are not part of the actual system dynamics, but are only used for model estimation.

$$\begin{bmatrix} \Delta\dot{q} \\ \Delta\dot{u} \\ \Delta\dot{w} \\ \Delta\dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{M_q}{I_{yy}} & \frac{M_u}{I_{yy}} & \frac{M_w}{I_{yy}} & 0 \\ \frac{X_q}{m} - w_0 & \frac{X_u}{m} & \frac{X_w}{m} & g\cos\theta_0 \\ \frac{Z_q}{m} + u_0 & \frac{Z_u}{m} & \frac{Z_w}{m} & g\sin\theta_0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta q \\ \Delta u \\ \Delta w \\ \Delta\theta \end{bmatrix} + \begin{bmatrix} \frac{M_{\delta_e}}{I_{yy}} & b_1 \\ \frac{X_{\delta_e}}{m} & b_2 \\ \frac{Z_{\delta_e}}{m} & b_3 \\ 0 & b_4 \end{bmatrix} \begin{bmatrix} \Delta\delta_e \\ 1 \end{bmatrix} \tag{1}$$

There exist 46 of such models[17] for different flight conditions. Very recently, efforts have been made to merge these local models into one global model using a Linear Parameter-Varying approach,[9] where the airspeed and angle of attack are used for scheduling. This model is used in some simulations in this research.

For the time-varying part of the model,[8] a third order Fourier series is used, as shown in Eq. 2, where the base frequency equals the flapping frequency. Coefficients are available to predict the pitching moment, axial force, and out-of-plane longitudinal force. Only one set of coefficients is used in this research.

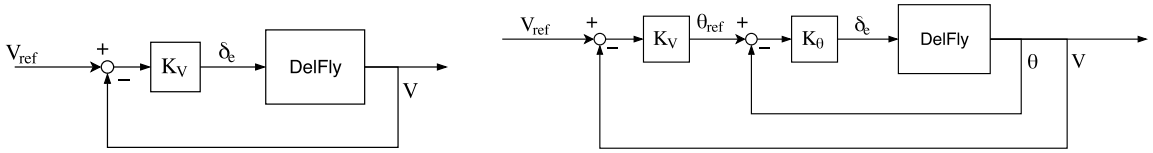
$$F_i(t) = \sum_{n=1}^{3} \left[a_n \sin(2\pi n f t) + b_n \cos(2\pi n f t) \right] \tag{2}$$

The full model is obtained by simply adding the partial models. The time-varying forces are scaled by their respective inertia terms and added to the accelerations in Eq. 1.

In this research, discrete-time models are used. A first order (Forward Euler) discretization is used, and the simulations are performed at 100 Hz. When flapping is introduced, however, the simulation time step is reduced to arrive at 500 Hz. The control algorithms are still run at 100 Hz to allow a fair comparison.

II.B. Control Objectives

The Reinforcement Learning algorithm can only learn if a single fixed task is repeated. The available models have only the elevator as an input, which is normally used for speed control[3] during hover and slow forward flight. Therefore, the natural control goal for this research is an airspeed controller. Altitude is not yet controlled, because accurate models accounting for throttle variations are not available. The task to be



(a) Single loop PI controller (PI)

(b) Cascaded PI controller (2PI)

Figure 2: Structures of the controllers used in this research.

performed by the airspeed controller is a 0.1 m/s increase in horizontal speed, when starting from a trimmed condition.

Proportional-Integral (PI) controllers are used for the underlying low-level controllers, because the integrating term facilitates trimming the DelFly. Two variants of the low-level controller are implemented, and shown in figure 2. The simplest, referred to as PI, takes as input the velocity tracking error, and processes this with a PI controller to calculate the required elevator deflection directly. The second variant uses two control loops. The velocity tracking error is fed to the outer PI controller, which specifies a pitch angle reference. The pitch angle tracking error is processed by the inner PI controller, which specifies the elevator deflection. This variant is referred to as cascaded PI, or 2PI.

Reinforcement Learning theory defines a *reward function*, which presents the controller with a reward on every time step. Within the framework of Markov Decision Processes, rewards can be based on the state and action taken, and may be probabilistic. For a velocity tracking controller, the most straightforward reward function is a quadratic punishment for deviations from the desired velocity. A punishment for control deflections may be added to limit the deflections.

If the goal of the controller is to maximize the cumulative reward, optimal controllers can be determined. Figure 3 shows the velocity V, pitch rate q, and elevator deflection δ_e for an example DelFly model with optimal controller for the reward function $U = -(V - V_{ref})^2$. It can be seen that the pitching motion is poorly damped. This seems to be due to the reward function, which only focuses on velocity deviations. If a penalty for q is added, better damping is obtained. The elevator deflection is sufficiently small, such that a punishment for control deflections is not required.

When flapping is introduced, there is a high-frequency vibration that cannot be fully suppressed by the controller. It was found that more favorable tracking is obtained when the controller is given knowledge of a filtered state. A fourth order Butterworth filter is used for this. Because this filter makes use of previous states, the control problem is not a Markov Decision Process. This may have implications when Reinforcement Learning methodology is applied to this system. Theoretical convergence analysis of the original RL algorithm in section III has only been performed[16] on Markov Decision Processes.

II.C. Simulation Methodology

A series of simulations is performed to assess the performance of the algorithms on the DelFly. The goal is to assess the abilities of the algorithms to address the five challenges. The first test considers only the variability: the algorithms are tested on the 46 local linear models that are available, without including flapping or disturbances in the simulation. Noise is fixed to the expected level, because zero noise would be unrealistic.

This tracking test is used as a baseline for a number of sensitivity analyses. The sensitivity to measurement noise is investigated by scaling the noise in steps. The L_2-norm of the vector of measurement uncertainties divided by Root-Mean-Square (RMS) values of the states is used as a noise measure. Next, the sensitivity to disturbances (i.e., process noise) is addressed, by superposing a (Gaussian) random elevator deflection at every time step. The standard deviation of this deflection is varied. During the disturbance test, noise is again fixed, in order to allow a fair comparison with the baseline.

A third sensitivity analysis considers the sensitivity to nonlinearities. An artificial nonlinearity, defined in the appendix, is introduced to fulfill this analysis.

The effect of the flapping motion is analyzed by starting from the variability test. The time-varying model part is superposed on the time-averaged part for each of the 46 models. The control algorithms need to be extended with the filter to deal with this.

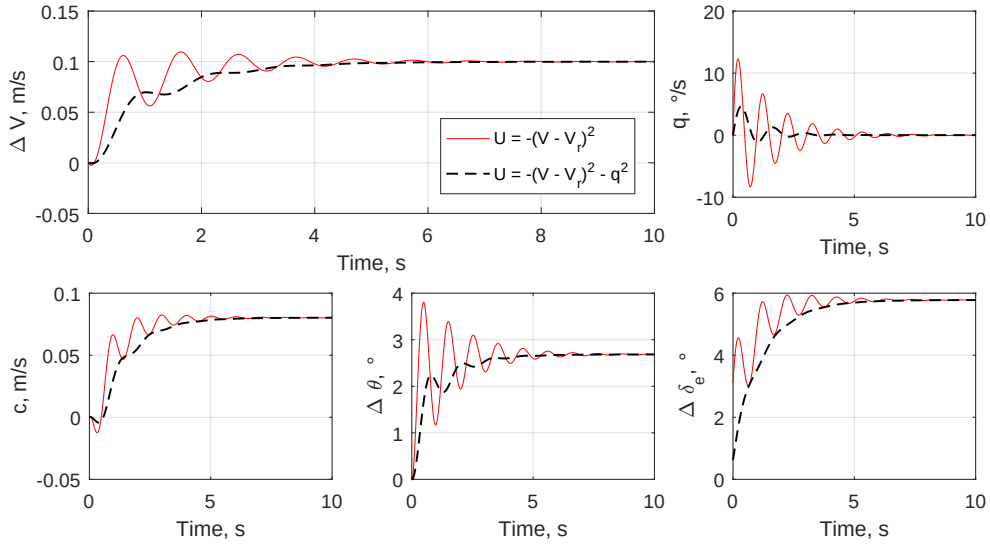


Figure 3: Effect of adding a pitch rate punishment on the optimal PI controller for an example system. The punishment improves damping, while still leading to effective tracking. Elevator deflections are sufficiently small in both cases.

A final analysis considers the recent global model of the DelFly. By introducing noise and flapping in the simulation, a test with this model can assess the performance of the control algorithms with respect to the challenges of noise, flapping and nonlinearity. Since there is only one such model available, assessing variability is not possible for the LPV model. Elevator disturbances are not introduced, since their expected magnitude is unknown, and the flapping motion is already a large disturbance.

The algorithms are assessed on their robustness to variability and tracking accuracy. The robustness to variability, i.e., safety, is defined by the fraction of controllers that is successful, i.e., obtains a cumulative reward higher than -100 (which is the reward obtained without any control) during every episode. Tracking accuracy is determined by the mean cumulative reward over ten episodes of all successful controllers. A combined score is obtained by giving all unsuccessful controllers a cumulative reward of -100, and averaging over all controllers.

III. Policy Gradient Algorithm

The first algorithm considered is a Reinforcement Learning algorithm of the actor-only category. This means that value functions, which are very common in RL, are not required. Instead, a parametrized actor is used to perform a repetitive task with a certain reward function. If the goal is to maximize cumulative reward, this is equivalent to a straightforward (stochastic) optimization problem, which can be solved using any optimization method. A gradient-based approach is used in this paper.

III.A. Using an inaccurate model

Determining the gradients is easier if a model is available during online control, because trial runs can be performed in simulation, while only the best settings are used in the real system. This reduces the number of policies that are evaluated online, and thereby reduces the probability of using a dangerous policy. However, the model has to be accurate for this to work correctly.

In this paper, an algorithm[16] is used which uses an inaccurate model to suggest parameter changes, and uses the real system to obtain the cumulative reward. Algorithm 1[16] shows an implementation of this Policy

Algorithm 1 Actor-only approach using inaccurate models[16]

Input: T, with a locally optimal parameter setting ψ_+
Output: ψ

repeat
  $\psi \leftarrow \psi_+$
  Execute the policy with ψ in the real system, and record the state-action trajectory $x(0)$, $u(0)$, $x(1)$, $u(1)$, ..., $x(n)$, $u(n)$
  for $t = 0, 1, 2, \ldots, n-1$ **do** {Construct a time-varying model $f_t(x,u)$ from $\hat{T}(x,u)$ and the trajectory}
    $f_t(x,u) \leftarrow \hat{T}(x,u) + \left[x(t+1) - \hat{T}(x(t),u(t))\right]$
  end for
  Use a local policy improvement algorithm on $f_t(x,u)$ to find a policy improvement direction d
  Determine the new parameters using $\psi_+ \leftarrow \psi + \alpha d$. Perform a line search in the real system to find α.
until ψ is not improved by the line search

Gradient (PG) method. The real system dynamics are represented by Eq. 3. This is approximated by an inaccurate model $\hat{T}$.

$$x(t+1) = T(x(t), u(t)) \tag{3}$$

This algorithm requires the construction of the time-varying model $f_t(x,u)$. $\hat{T}(x,u)$ provides a time-invariant representation of the system, which will (in general) not match the experienced trajectory exactly. By including the time-varying bias term $\left[x(t+1) - \hat{T}(x(t),u(t))\right]$ at every time step, $f_t(x,u)$ will match the experienced trajectory. The gradient obtained from $f_t(x,u)$ is then the gradient on the experienced trajectory.

For this algorithm, convergence to a local optimum in a Markov Decision Process can be proven if the local improvement algorithm is policy gradient.[16] However, this is only applicable if the true dynamics $T(x,u)$ are deterministic. Policy gradient algorithms similar to this one have been flight-proven using optical tracking systems,[6,18] which points towards suitability of the algorithm for the flight vehicle under investigation.

III.B. Variable learning rate

Performing a line search in the real system, as suggested in algorithm 1, results in a large number of policy evaluations. Furthermore, taking a step that is too large may result in unstable controller. A new algorithm with a virtual line search procedure is proposed in this paper. This is presented in algorithm 2.

This algorithm avoids time-consuming line searches by using a variable step size. An optimal step is size is found in the virtual system, and this step size is multiplied by a factor β for use in the real system. The learning rate β is decreased when the performance deteriorates. It has only one tunable parameter, which is the initial learning rate. In this research, the parameter step size is limited in order to increase safety.

When a flapping motion is present, the algorithm is adapted by consistently feeding the filtered state and reward to the controller. The unfiltered control deflection is used, because the deflection is based on the filtered state. The internal representation is left unchanged, i.e., the controller does not have the information that the state is filtered.

III.C. Controller behavior

An example of the behavior of the PG controller is shown in figure 4. Because of the actor-only approach used, the controller can only improve its gains once per episode. Nevertheless, it quickly reaches its optimal gains: the gains hardly change after four episodes. This controller is initialized with $\beta = 0.8$, and therefore gets close to the optimum after a single episode. The gains are fine-tuned during the later episodes.

The plots of airspeed and pitch rate reveal that the rise time decreases slightly, at the expense of a lower damping. When looking at the elevator deflection, it is found that the deflection signal is slow. The controller is not actively counteracting the oscillations, but relies on the natural damping of the vehicle. A different controller structure, such as the cascaded PI controller, is required for active damping.

Algorithm 2 Policy Gradient algorithm used in this research. This is a variation to algorithm 1.

Input: T, with an initial parameter setting ψ_+, and initial learning rate β
Output: ψ

Set $J_\psi \leftarrow -\infty$
repeat
  Execute the policy with ψ_+ in the real system, and record the state-action trajectory $x(0)$, $u(0)$, $x(1)$, $u(1)$, ..., $x(n)$, $u(n)$, with associated cumulative reward J_{ψ_+}
  if ψ_+ improves over ψ **then**
    $\psi \leftarrow \psi_+$
    for $t = 0, 1, 2, \ldots, n-1$ **do** {Construct a time-varying model $f_t(x,u)$ from $\hat{T}(x,u)$ and the trajectory}
      $f_t(x,u) \leftarrow \hat{T}(x,u) + \left[x(t+1) - \hat{T}(x(t),u(t))\right]$
    end for
    Use a numerical algorithm on $f_t(x,u)$ to find the gradient $\nabla_\psi J$
    Determine new parameters using $\psi_- \leftarrow \psi + \alpha\nabla_\psi J$. Perform a line search in $f_t(x,u)$ to find the optimal α.
    Take a partial step towards ψ_- using $\psi_+ \leftarrow \psi + \beta\alpha\nabla_\psi J$
  else
    Halve the learning rate $\beta \leftarrow 0.5\beta$
    Take a smaller step towards ψ_- using $\psi_+ \leftarrow \psi + \beta\alpha\nabla_\psi J$
  end if
until steps are sufficiently small

IV. Classification Algorithm for Machine Learning Control

Because the Policy Gradient approach requires one full episode with the initial controller, it can never achieve a lower crash rate than a fixed PI controller. If greater tolerance to variability is required, the controller should be able to change its gains rapidly during an episode. Conventional adaptive Reinforcement Learning algorithms, such as Heuristic Dynamic Programming,[10] are expected to be unable to adapt fast enough to recover an initially unstable controller.

The task of controlling an unknown DelFly draws parallels with the 2008 and 2009 RL Competitions involving helicopter hovering. In this competition, the task was to control a helicopter with unknown dynamics.[19] The 2008 competition was won by a high-level controller selection approach of RL,[19] that requires all of N predefined controllers to be attempted on the real system, and selects the best one. This comes with the risk of attempting an unstable controller and crashing the system. Only safe controllers can be used, which are able to stabilize most DelFly systems. It is expected that the variability between different DelFly's makes it difficult to use the controller selection algorithm effectively.

A later algorithm for the RL Competitions uses a stochastic model in combination with Model Predictive Control.[20] This leads to accurate control and allows rapid changes in control parameters, but is computationally prohibitive.

This paper proposes a new approach, which combines the strengths of the controller selection approach and the stochastic Model Predictive Control approach. The basis of this approach is that an identified model, even if it has a large uncertainty, can be used to predict the effectiveness of the predefined controllers. Instead of performing time-consuming simulations to optimize a policy, the model is used directly to select one of the predefined controllers.

Figure 5 demonstrates the concept of this controller, named Classification Algorithm for Machine Learning control (CAML). A recursive model identification algorithm is used to identify the model online. Also, the parameter (co)variance of the model is identified, in order to obtain a measure of the uncertainty in the model.

The most straightforward approach with knowledge of the model and the parameter variance is to perform simulations with all available controllers for a wide range of models, whose distribution is defined by the identified mean model and covariance. However, performing these computations online is considered computationally prohibitive. Instead, a Neural Network (NN) is trained on a dataset of simulations that were

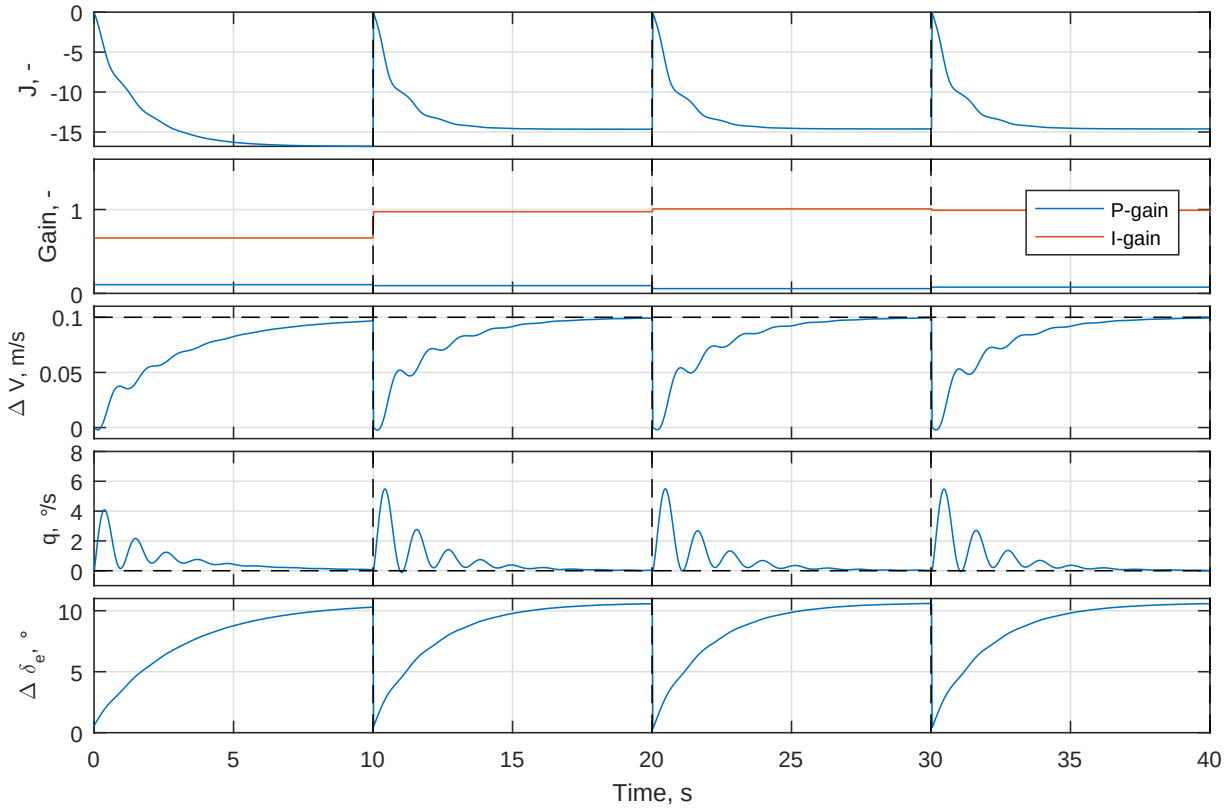


Figure 4: Example behavior of the Policy Gradient controller with PI structure when tracking a 0.1 m/s airspeed step, over four episodes. The increasing cumulative reward per episode (J) shows that the gradient-based learning procedure typically results in increasing performance over time, with changes once per episode.

performed offline. This NN takes the model and parameter variance as inputs[b], and outputs a prediction of the most suitable controller. This is a classification problem, where the combination of model and parameter variance must be allocated to one of N gain set categories. The softmax layer outputs N numbers, where each number represents the probability p_i that the input belongs to the corresponding gain set category. The gain set with the highest probability is selected on every time step.

CAML is intended to improve safety by allowing the gains to change rapidly. Upon a switch of gain set, the integral term is reset in such a way that the commanded elevator deflection remains constant. The CAML algorithm has a weak inherent protection against instability: if an unstable gain set is selected, the deviations from the trimmed state will become large. The inputs to the online model identification algorithm become large, so the identified model will change rapidly. If the new model is fed to the NN, it may select a different gain set. This mechanism does not result in guaranteed safety, but simulations show that it effectively improves safety.

Conventional gain scheduling methods predefine different linear controllers for a nonlinear system, where the state is used to switch between controllers. CAML can also be used for nonlinear systems, by identifying a linear approximation, on which the gain selection is based. However, CAML performs these computations during operation. This makes the algorithm suitable for unknown systems within the space of expected systems, whereas gain scheduling controllers are only suitable for the system they are designed for.

IV.A. Predefined gain set generation

The predefined gains used by CAML are tuned by examining the 46 known models. For every model, an optimal PI gain set is determined by means of an offline optimization routine (gradient descent with multiple

[b]The computational complexity of CAML is dependent on the number of states. This research only considers four-state models. A formal analysis is required to find a practical limit to the number of states.

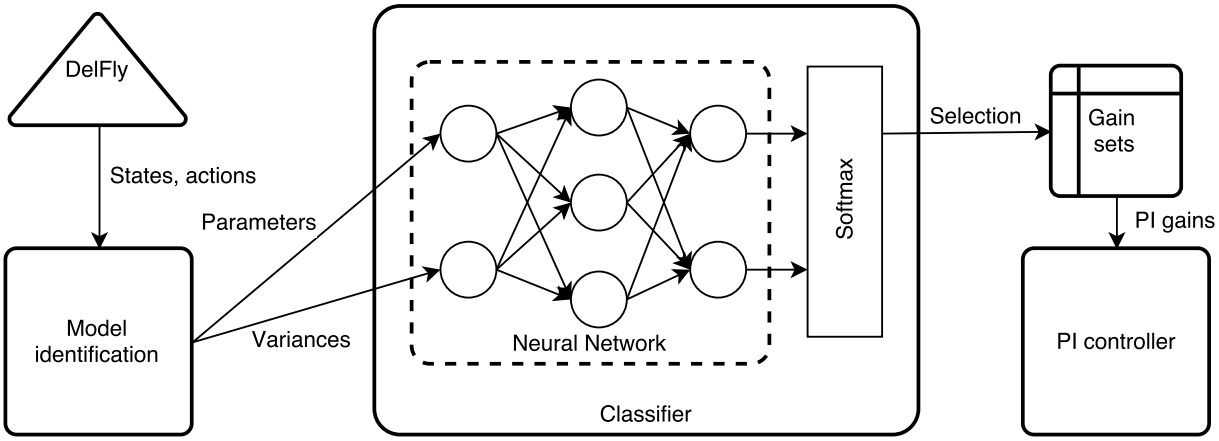
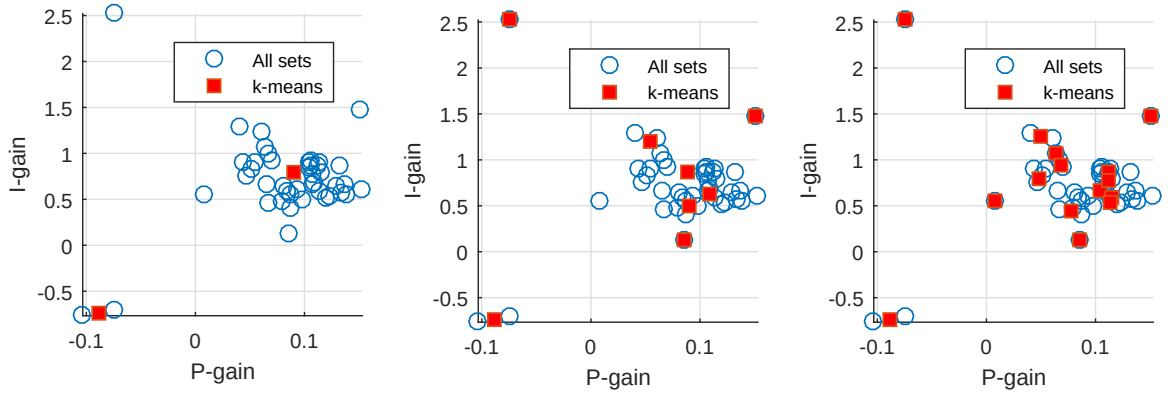


Figure 5: Principle of CAML. The parameters and variances of a model, identified online, are fed to a classification Neural Network, which selects the most appropriate gain set for the low-level controller.



(a) $k = 2$ (b) $k = 8$ (c) $k = 15$

Figure 6: Optimal PI controllers for the known models. The k-means clustering algorithm makes a smaller selection. It captures the outliers well, but cannot fully match the spread in the main cloud.

starts). These gains optimize the cumulative reward as defined in section II. The closed-loop stability of the resulting 46 PI controllers with their models are analyzed, and unstable gain sets are discarded.

The resulting gain sets[c] are shown in figure 6. For the selectable gain table, it is desired to have less than 46 options. A k-means clustering algorithm (k-means++[21]) is used to obtain a smaller representative set of gains. Figure 6 shows the resulting gain sets for the CAML controller for several k. The same process is followed for the cascaded PI controllers.

IV.B. Training the Neural Network

The classifier must be trained using a large number of examples. These examples are sets of model parameters, variances, and correct selections. The pairs of model parameters and variances are generated with the 46 available models. For every combination of models A and B, algorithm 3 is used. This yields series of aerodynamic coefficients starting at model B, with suitable initial variance[d], and slowly converging to model A with lower variance. The result is a set of realistic combinations of model parameters and variances.

For a deterministic set of training models, the training targets are determined by simulating all N_c gains

[c]There are two models with inverted control effectiveness, which results in negative optimal gains. The model with negative P-gain, but positive I-gain has normal control effectiveness, but is very different from the remaining models.

[d]The initial model variance is based on the variance in the aerodynamic coefficients of the 46 models.

Algorithm 3 Generation of pairs of model parameters and variances

Input: 3-2-1-1 response of DelFly A, with n data points $x_A(t), u_A(t)$
Output: An array V of pairs of model parameters and variances
1: Initialize a model M at the aerodynamic coefficients of DelFly B
2: **for** $t = 0, 1, 2, \ldots, n-1$ **do**
3: Update M using Recursive Least Squares, with data point $x_A(t), u_A(t) \rightarrow x_A(t+1)$
4: Save the parameters and variances of M to $W(t)$
5: **end for**
6: Take random samples of W, resulting in a smaller array V

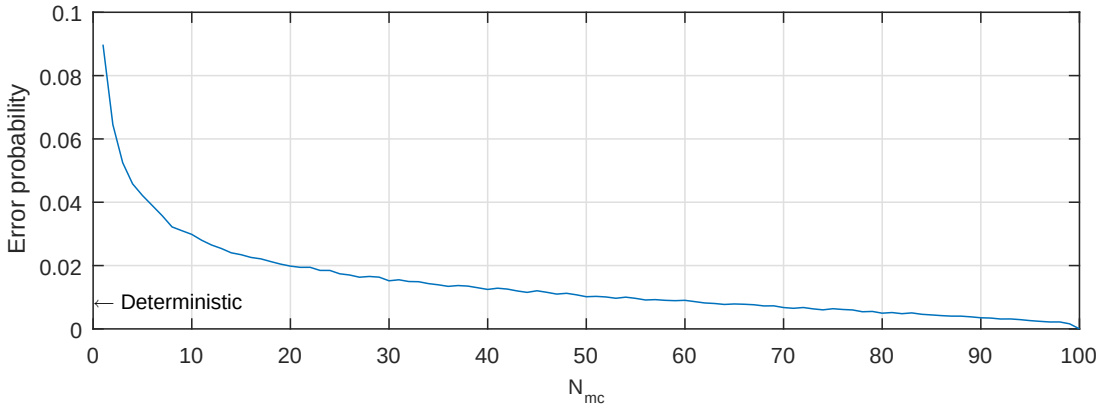


Figure 7: Error probability in the training data as a function of the number of random models. The best gain set when testing with $N_{mc} = 100$ is considered perfect. The low error rate of 0.009 if variance is disregarded indicates that a variance input is not necessary.

sets with all N_t models. The twelve important aerodynamic coefficients are the training inputs[e], and the training targets are the indices of the most suitable controllers. Training is performed using scaled conjugate gradient training with early stopping (80% training data, 20% validation data). Five independent NNs are trained, and the best performing one in terms of error rate on the complete dataset is selected.

An additional step is required to account for a nonzero variance. For each training model, N_{mc} new, random models are created, by adding Gaussian random values (scaled by the corresponding variance) to each of the aerodynamic coefficients. The optimal controller is determined by averaging the performance of each gain set over the N_{mc} random models[f]. In this case, the training input consists of twelve aerodynamic parameters and twelve variances.

The variance input to the NN ensures that the controller can select a safe gain set when the uncertainty is high. However, it adds twelve input dimensions to the NN and makes training more time-consuming, by a factor N_{mc}. The number of random models should be high enough, to avoid incorrect gain selections in the training data.

Figure 7 is used to analyze the appropriate value of N_{mc}. In this figure, it is assumed that selection is perfect with $N_{mc} = 100$[g], i.e., 100 random models related to each training model. This is repeated for 1000 training models, and the error probability in the training data with lower N_{mc} is analyzed. For comparison, it is found that the error probability is 0.009 if the variance is disregarded, and only the identified model is analyzed. In this particular analysis, at least 57 random models must be used to match this. The order of magnitude of the error rate achievable by the NNs on such datasets is 0.05. It is therefore decided that the variance input does not contribute effectively, and only NNs without variance input are considered further in this research.

[e]A complication is that the A-matrices of the models include both aerodynamics coefficients, which are inputs of the NN, and kinematic terms. The training simulations are performed using models with different aerodynamic coefficients, but equal kinematic terms, corresponding to a trimmed airspeed of approximately 0.8 m/s.

[f]If a gain set is unstable for a random model, a fixed penalty is given, in order to avoid excessive punishment of unstable models.

[g]This makes this analysis unreliable for N_{mc} close to 100.

The training models with variance are identified by letting Recursive Least Squares converge from one model to another. This is a rather limited procedure of random model generation, because the resulting models will always be close to the 46 known models. A more general procedure is to generate random training models, based on the means and variances of the aerodynamic coefficients of the 46 known models. For each coefficient, the mean and variance among the 46 models is determined. An independent Gaussian distribution for every parameter is assumed to generate new models. This is performed after repeatedly removing outlier models[h], which have an aerodynamic coefficient varying more than 3σ from the mean. The resulting training dataset covers a larger space of possible models, but NN training is more difficult, with resulting error rates in the order of 0.15. It is found that both methods of training model generation result in similar performance of the CAML controller when tested on the 46 known models.

IV.C. Model Identification

This method is highly dependent on the identification of a model of the dynamics of the DelFly. This section describes how this model is identified during operation. The model structure in Eq. 1 is used. For this application, it is more convenient to use the discrete-time model in Eq. 4, which is obtained by means of a first order (Forward Euler) discretization. Note that several matrix elements are set to zero, and some elements are dependent on the initial state only. Also, the last column of the input matrix does not contribute to the dynamics, but only to the trimmed condition. This results in twelve important aerodynamic parameters to be identified.

$$\begin{bmatrix} q(t+1) \\ \mathfrak{u}(t+1) \\ w(t+1) \\ \theta(t+1) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} - w_0\Delta t & a_{22} & a_{23} & g\cos\theta_0\Delta t \\ a_{31} + u_0\Delta t & a_{32} & a_{33} & g\sin\theta_0\Delta t \\ \Delta t & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q(t) \\ \mathfrak{u}(t) \\ w(t) \\ \theta(t) \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ 0 & b_{42} \end{bmatrix} \begin{bmatrix} \delta_e(t) \\ 1 \end{bmatrix} \tag{4}$$

The online model identification problem is to determine the parameters of the matrices A and B from flight data, such that the model error $\hat{e}$ is minimized over time. This is done separately for every row of these matrices. As an example, Eq. 5 shows the (noise-free) identification problem for the pitch rate row.

$$q(t+1) = a_q(t)w_q + \hat{e}_q(t) = \begin{bmatrix} q(t) & u(t) & w(t) & \delta_e(t) & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & b_{11} & b_{12} \end{bmatrix}^T + \hat{e}_q(t) \tag{5}$$

An online model identification algorithm selects the weight vectors w such that the model errors $\hat{e}$ are minimized over time. The most straightforward method of online model identification is the Recursive Least Squares (RLS) algorithm.[22] This algorithm recursively solves for the parameters w in order to minimize measurement error e in problems of the form of Eq. 6. If $f(y,u)$ is linear, this problem is a special case of the auto-regressive moving-average model with exogenous input (ARMAX), which is shown in Eq. 7.

$$y(t) = \left[f(y(t-1), u(t-1))\right]^T w + e(t) \tag{6}$$

$$y(t) = \begin{bmatrix} -y(t-1) & \cdots - y(t-n_a) & u(t-1) & \cdots & u(t-n_b) & e(t-1) & \cdots & e(t-n_c) \end{bmatrix} w \tag{7}$$

The model identification problem can be seen as a vector form of the ARMAX problem, where the state vector $x(t)$ has to be predicted by the model. If there is a measurement noise vector $E(t)$, this problem can be written as in Eq. 8, where the measured state is denoted by $z(t) = x(t) + E(t)$.

$$\begin{aligned} x(t) &= Ax(t-1) + Bu(t-1) \\ z(t) &= Az(t-1) + Bu(t-1) - AE(t-1) + E(t) \end{aligned} \tag{8}$$

Because there is measurement noise at both t and $t-1$, the underlying assumptions of the RLS estimator are violated. If RLS is used anyways (implicitly assuming that $AE(t-1) = 0$), the result will be inaccurate.

[h] This is required, because these outlier models cause unrealistically large variations among the newly generated random models. A disadvantage is that the trained classifier will be unable to perform accurate predictions for the outlier models.

The Recursive Maximum Likelihood algorithm RML1,[22,23] also called Extended Least Squares, is an option to solve this problem. This algorithm extends the regressor in RLS with the previous measurement error $e(t-1)$. Since this measurement error is not known, the prediction error $\hat{e}(t-1)$ is used instead. It is straightforward to extend this to the vector case.

Due to the structure of Eq. 8, the correct weights on $\hat{E}(t-1)$ and $z(t-1)$ are directly related, and can be constrained. Since $\hat{E} = z - \hat{x}$, this is equivalent to using the predicted previous state $\hat{x}(t-1)$ instead of $x(t-1)$ in the conventional RLS estimator. The resulting recursive algorithm is shown in Eqs. 9 to 13, where the regressor $a_i(x,u)$ is an affine vector function of the state and input (like $a_q(t)$ in Eq. 5).

$$r_i(t) = z_i(t) - w_i^T(t-1)a_i(\hat{x}(t-1), u(t-1)) \tag{9}$$

$$L_i(t) = \frac{P_i(t-1)a_i\left(\hat{x}(t-1), u(t-1)\right)}{a_i^T\left(\hat{x-1}(t), u(t-1)\right)P_i(t-1)a_i\left(\hat{x}(t-1), u(t-1)\right) + \lambda} \tag{10}$$

$$w_i(t) = w_i(t-1) + L_i(t)r_i(t) \tag{11}$$

$$P_i(t) = \frac{1}{\lambda}\left[P_i(t-1) - L_i(t)a_i\left(\hat{x}(t-1), u(t-1)\right)P_i(t-1)\right] \tag{12}$$

$$\hat{x}(t) = w_i^t(t)a_i\left(\hat{x}(t-1), u(t-1)\right) \tag{13}$$

Identification of the time-averaged model part was originally done by filtering with a fourth order Butterworth filter.[8] The non-causal zero-phase variant of this filter was used,[8] but this is not possible during online identification. This paper uses the causal variant of the Butterworth filter, which adds a phase shift, affecting model identification. Implementing the filter in the RML1 method is non-trivial, because the previous state $\hat{x}(t-1)$ is internal, while the current state measurement $z(t)$ is external. It is found that the conventional RLS algorithm is more resistant to this filtering effect than RML1. In this research, RML1 is used for all simulations without filtering, whereas RLS is used for simulations with filtering.

IV.D. Algorithm parameter selection

There are three hyperparameters governing the behavior of the classification controller: the number of selectable gain sets, the number of neurons in the hidden layer, and the number of training models. The optima of these parameters are clearly interrelated. Also, it needs to be considered that the gain sets and training models are based on the models that are also used for testing. This will give problems if the previously mentioned numbers are set too high.

Due to the time-consuming nature of the computations, a three-dimensional optimization on a fine grid is infeasible. Rather than optimizing on a coarser grid, it is opted to perform manual optimization, using engineering judgment. A sensitivity analysis on the selected near-optimal point is performed for every parameter individually. The baseline tracking test is used for tuning the parameters. All 46 models are used for optimization, because the number of models will also influence the optimal settings. Later, it is tested if splitting the data into a training set and a test set gives significantly different results.

IV.D.1. Number of gain sets

The number of gain sets in the controller should be high enough to cover the space of desirable gain combinations. A higher number should lead to higher possible performance. If the number becomes too large, however, the neural network will have problems making a correct selection, and the performance will decrease. While holding the numbers of neurons and training models fixed, figure 8 shows the performance for different numbers of gain sets.

The figure does not show the expected trend of performance with the number of gain sets. This is a first indication that CAML is not working optimally on this problem. Figure 6 provides an explanation for this behavior. If only two gain sets are used, CAML nearly always picks the gain in the middle of the main cloud in figure 6a. When then number of gain sets is increased, more outliers are captured, but the main cloud is not fully covered. It is found that fixed PI controllers with gains near the center of the cloud all lead to similar scores. Thus, increasing k will lead to more outliers in the gain sets, and more comparable gains. The latter consequence will tend to increase the score, but the first consequence will increase the risk of incorrect gain selection. It is expected that higher performance would be found if the gain sets are spread more evenly over the main cloud.

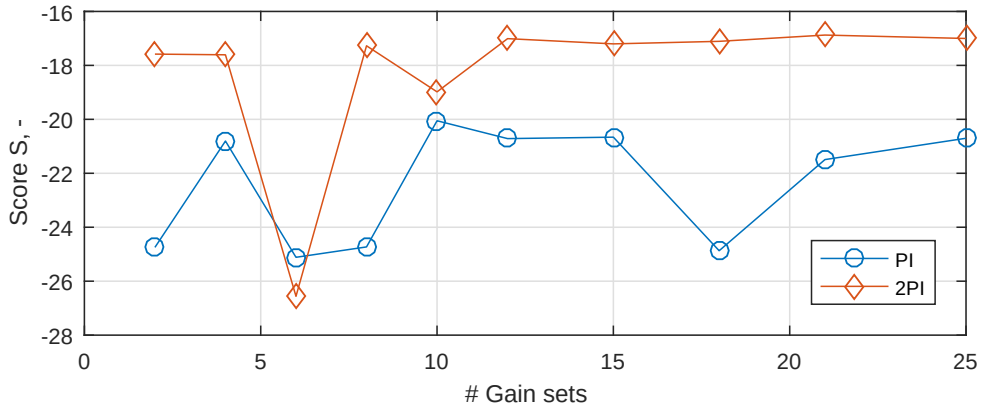


Figure 8: Tuning of the number of gain sets. Unexpected drops in performance are found for some numbers. Fifteen controllers is regarded near-optimal for both single and double loop PI controllers.

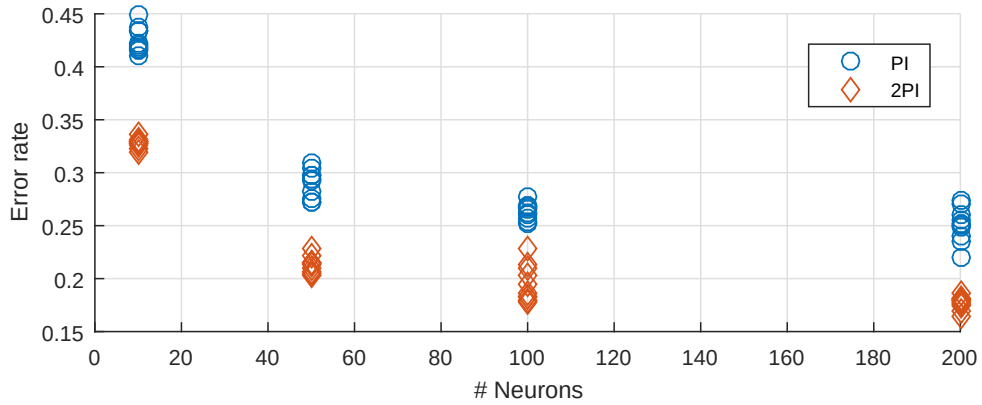


Figure 9: Tuning of the number of gain sets. Overfitting does not seem to be a problem. A number of 100 neurons is considered sufficient.

IV.D.2. Number of neurons

The number of neurons must be large enough to capture the complexity in the training data, but Neural Networks that are too large suffer from overfitting, and require longer computation times. The optimal number of neurons with respect to overfitting is analyzed by generating a new set of 10000 testing models. Neural networks of various sizes are trained on the training data, and tested on the test data. This is repeated ten times, as shown in figure 9.

It can be seen that the error rate is steadily decreasing for the numbers of neurons considered. Since the test is performed on a separate dataset, this is an indication that overfitting is not yet problematic for these numbers of neurons. The error rates appear to stabilize at 100 neurons. Increasing the number of neurons further than 100 is not worth the computational time, so a number of 100 neurons is selected for both single and double loop controllers.

IV.D.3. Number of training models

Increasing the number of neurons will always result in increased performance, because more training data allows larger neural networks without overfitting, and thereby a more powerful classifier. However, the (offline) computational time limits the number of training models that is feasible in this research. Figure 10 analyzes the error rates on the 10000 testing models for different numbers of training models.

As expected, the performance keeps decreasing for larger numbers of training models. For high numbers, the error rate seems to stabilize. It is decided that 40000 models is sufficient for the purposes of this research.

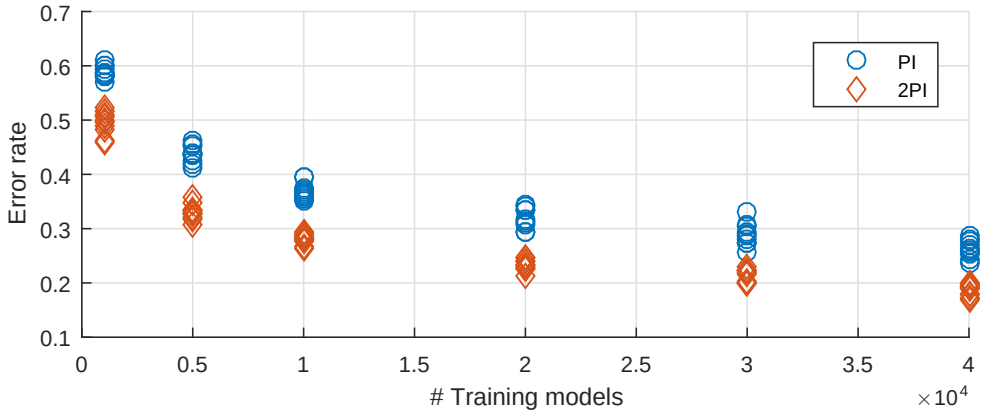


Figure 10: Tuning of the number of training models. It is decided that 40000 models is sufficient.

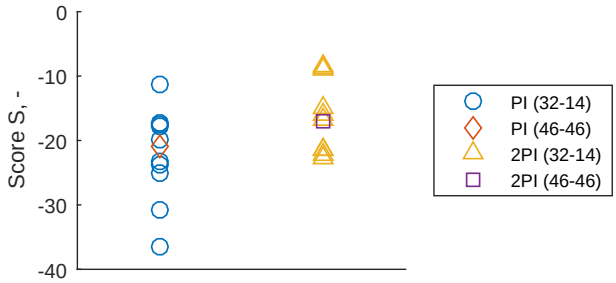


Figure 11: Variability results if 32 of the models are used for training, and 14 for testing, repeated ten times with random divisions. The controller trained and tested on the full dataset does not appear to be overly optimistic.

IV.D.4. Internal validation

It is common to split the data into training data and test data when testing tuned controllers, but this would limit the power of the classification controller. The allowable numbers of gain sets and neurons would have to be decreased. In this research, the tuning and testing are both performed on the full dataset. In an additional analysis, the data is split up several times, to investigate if the results with the full dataset are overly optimistic.

In this additional analysis, 70% (32) of the 46 models are used for tuning and training, and the remaining 14 are used for testing. The numbers of gain sets, neurons, and training models are left at their previously optimal values. The models in the training set are used to generate 40000 random models (disregarding the same outliers as before) on which new neural networks are trained, and to select appropriate gain sets. This is repeated ten times, with new divisions into training and test data. The resulting scores are shown in figure 11.

It is found that the controller trained and tested on the same models is not unreasonably optimistic. There is no reason to assume that using the same models for training and testing is causing any problems.

IV.E. Controller behavior

Figure 12 shows an example of the behavior of the CAML controller with a PI structure. The example system is the same as in figure 4, and the figures can be compared directly. During the first seconds, the model, which is at the input of the NN, is uncertain and changes rapidly. This causes fast variations in the outputs of the NN, which are the softmax probabilities p_i for each gain set, which is reflected by the varying gains. The number of gain switches decreases during later episodes, due to the slower changes in the model. Gain switches are concentrated in the first seconds of every episode, because the excitation of the system is largest in these periods.

A disadvantage of the CAML algorithm is that the performance does not necessarily improve over time.

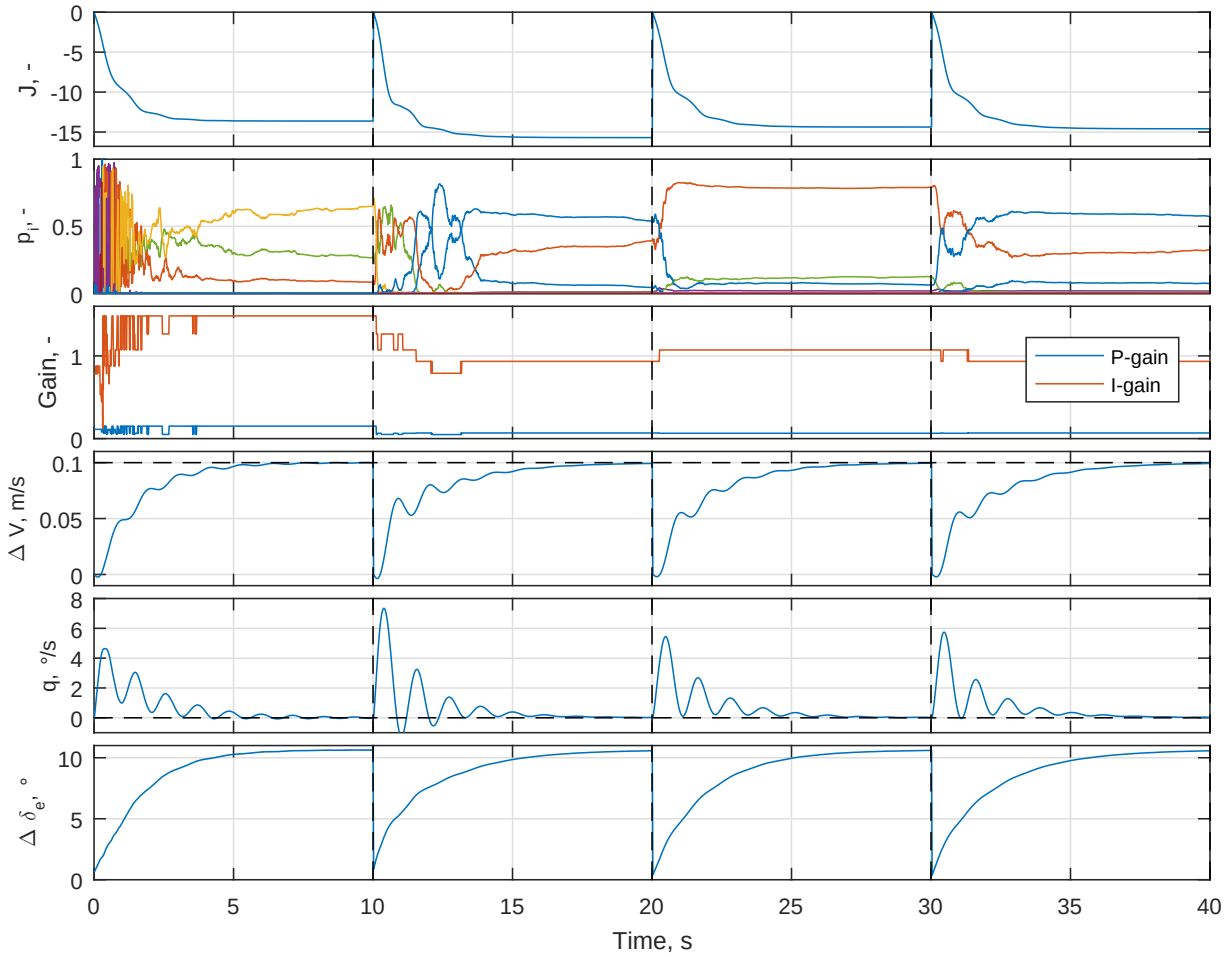


Figure 12: Example behavior of the CAML controller with PI structure when tracking a 0.1 m/s airspeed step, over four episodes. The softmax output probabilities (p_i) of each gain set, and therefore the gains themselves, change rapidly, especially during the beginning of each episode. Changes occur less often during later episodes. Performance, measured with J, does not necessarily improve over time.

The identified model improves, but the NN classifier has a nonzero error rate, and can therefore select an inappropriate gain set even if the perfect model is known. In spite of this limitation, CAML has a tendency to improve over time. Note that the gains selected during the final episodes in figure 12 are close to the optimal gains found by the Policy Gradient controller in figure 4.

V. Results and Discussion

This section treats the results of the test problems mentioned in section II. Results are shown for the Policy Gradient algorithm and for CAML. Additionally, results for a fixed PI controller are shown. All gain sets from the CAML database are attempted in a fixed PI controller, and the best performing set is shown.

V.A. Baseline tracking test

Figure 13 shows the performance of the algorithms during the baseline tracking test, where the goal of the controller is to track a 0.1 m/s velocity step on all of the 46 available models. The score S is defined in Eq. 14. Wilcoxon's signed rank test[24] is used to assess the statistical significance of the differences in tracking

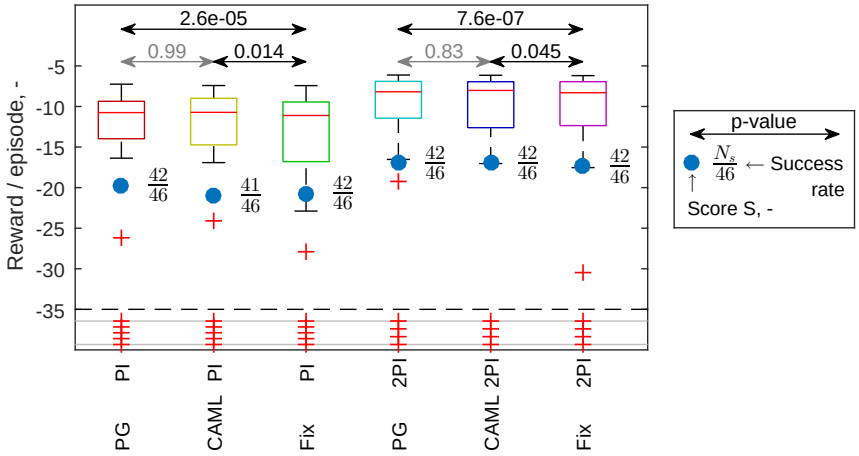


Figure 13: Results of the baseline tracking test. The statistical significance is shown for some differences in tracking accuracy. The Machine Learning algorithms perform better overall.

accuracy[i], rather than score. This test works in a pairwise setting; the p-values of several pairs are shown in figure 13. The difference is considered significant if $p < 0.05$.

$$S = -100 \cdot N_{unstable} + \sum_{i=1}^{N_{stable}} J_i, \qquad J = \sum_t U(t) = -\sum_t \left[(V(t) - 0.1)^2 + q(t)^2\right] \tag{14}$$

It can be seen that all algorithms manage to stabilize at least 41 out of 46 models. When inspecting the single loop PI variants of the ML algorithms, the score of the PG algorithm is slightly higher, but the increase in tracking accuracy is not significant. Both algorithms outperform the fixed PI controller in terms of tracking accuracy. The initial gains of the PG algorithm are the same as the gains of the fixed PI controller. The fact that the success rates are equal, shows that the variable learning rate procedure is effective in avoiding instability.

The single loop variant of CAML has a lower success rate than the two other approaches. Further inspection shows that four of the 46 linear DelFly models cannot be controlled successfully by a single loop PI controller of the given structure. Furthermore, there is a universal set of gains, which results in success for the 42 remaining models. By using this gain set as the initial setting, this allows the PG algorithm to successfully control those 42 models. CAML cannot improve safety with its rapid gain changes, because changes from the universal gain set will not allow control of the four 'uncontrollable' models. It therefore only experiences its disadvantage, which is an occasionally incorrect gain selection. This results in instability of one of the 42 controllable models.

For cascaded PI controllers, the performance of both ML algorithms looks comparable to the performance of the fixed controller in figure 13. However, the tracking accuracy of the fixed controller is actually significantly worse, due to the distribution of the outliers in the figure. Cascaded PI controllers lead to higher performance than single loop PI controllers, for all algorithms considered. All algorithms allow control of 42 out of 46 models. Just as with the single loop variant, there is a universal gain set for those 46 models. Four models (the same four as before) cannot be controlled successfully by cascaded PI controllers.

V.B. Noise

The robustness of the algorithms to measurement noise on the states is presented in figure 14. Overall, it seems that increasing noise does not lead to a dramatic decrease in performance. The expected noise level of the DelFly does not lead to problems.

CAML is more sensitive to noise than the PG algorithm. This can be explained by looking at the way in which noise affects the algorithms. In the PG algorithm, measurement noise changes the time-varying part of

[i]This test assumes a zero hypothesis that all data comes from the same type of continuous distribution, which is clearly false if unstable models are included. All models with one or more unstable controllers are therefore disregarded in this statistical test. This means that the test only considers tracking accuracy, not the combined score S.

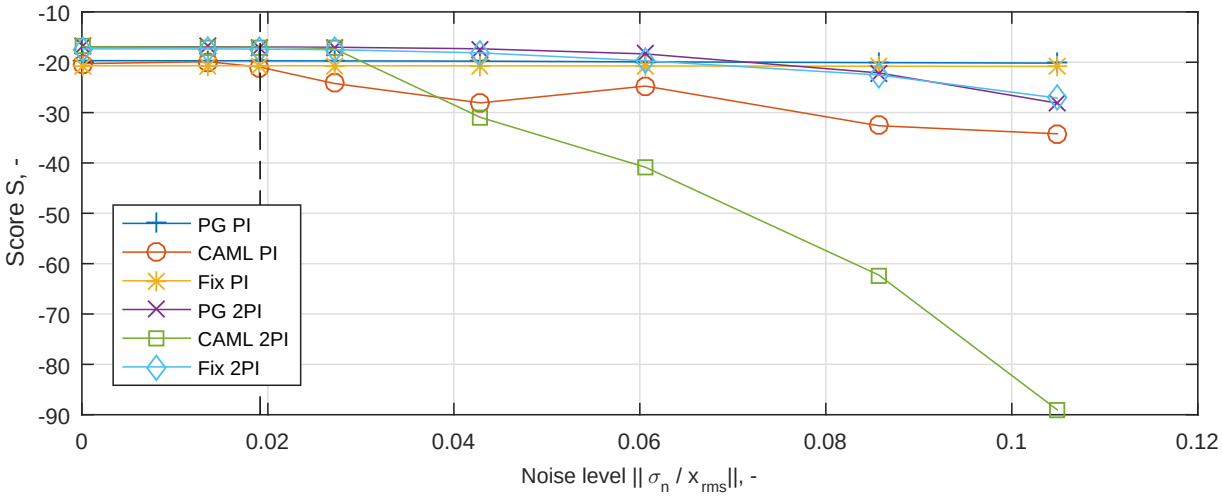


Figure 14: Sensitivity of the algorithms to noise. The PG algorithm is more robust to noise, with the single loop variant performing better.

the internal model, leading to a disturbance signal in the virtual gradient simulation. Since this disturbance is random, its effect is diminished by using longer measurement times. For CAML, noise gives problems with model identification. As explained in section IV, Recursive Least Squares identification cannot deal with noisy measurements of the current state. The RML1 algorithm solves this problem to some extent, but increased noise will still affect the identified model.

Another noteworthy conclusion is that the algorithms with cascaded PI controllers are more sensitive to measurement noise than the algorithms with single loop PI controllers. This seems to be due to the low-level control, because the fixed PI controllers show the same effect. Because cascaded controllers make use of two states, the noise on both states enters the PI controller, lowering the accuracy.

V.C. Disturbances

An unknown disturbance on the elevator will strongly affect the performance of the low-level PI controllers. Also, the algorithms will perform suboptimally. It is expected that the algorithms can deal with disturbances more easily than with noise, because the disturbance actually changes the state. The PG algorithm will sample the disturbance on each step, and will find the correct gradient for a simulation with this disturbance sequence. The model identification algorithm in CAML can deal with disturbances directly, since they result in an actual state change.

Figure 15 shows the performance of the algorithms with a varying level of disturbances. The robustness of the PG algorithm is found to be stronger than the robustness of CAML, because the algorithm can actively deal with the disturbances by moving to a more appropriate gain setting. CAML lacks this option.

In this test, it is found that cascaded PI controllers result in better scores than single loop PI controllers. This makes sense, because the double loop controllers can counteract disturbances on two states. An offset in pitch angle will eventually lead to a speed change (which is punished), and only the double loop controllers can counteract this.

V.D. Nonlinearity

When nonlinearities are introduced, the linear controllers will become less effective. Figure 16 shows that the nonlinearities considered are mild enough to allow accurate control with fixed PI controllers. The nonlinearity metric ϕ is defined in the appendix.

This analysis shows that the PG algorithm is more robust to nonlinearities than CAML. This was expected, because CAML requires identification of a linear model, which may be inaccurate on a nonlinear system. For the Policy Gradient approach, the nonlinearity is dealt with like a linear model deviation.

Double loop PI controllers are able to achieve higher performance than single loop controllers. The explanation for this is that cascaded controllers limit oscillations of the pitch angle, and thereby keep the

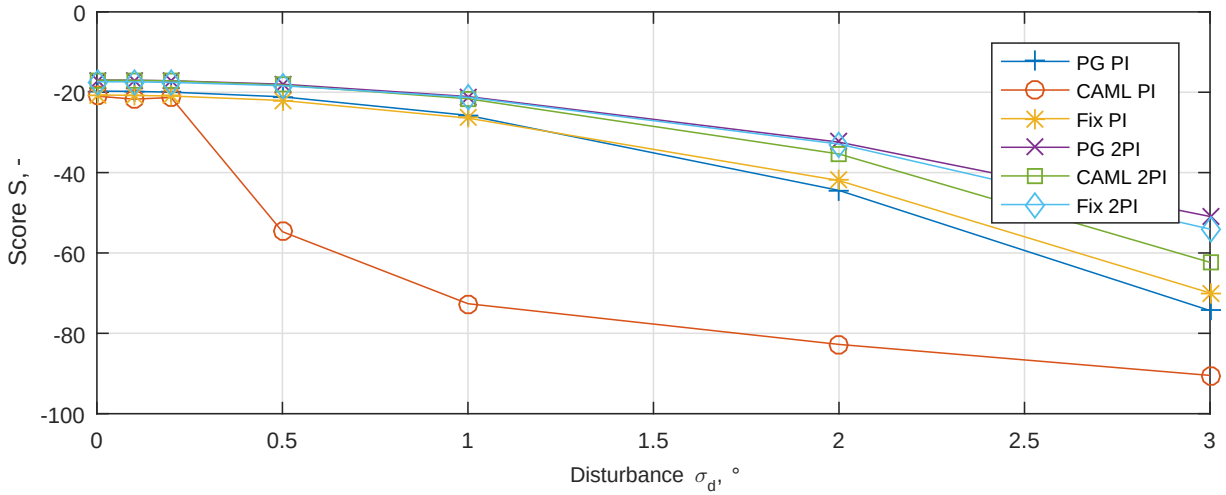


Figure 15: Sensitivity of the algorithms to disturbances. The PG algorithm is more robust to noise, with the double loop variant performing better.

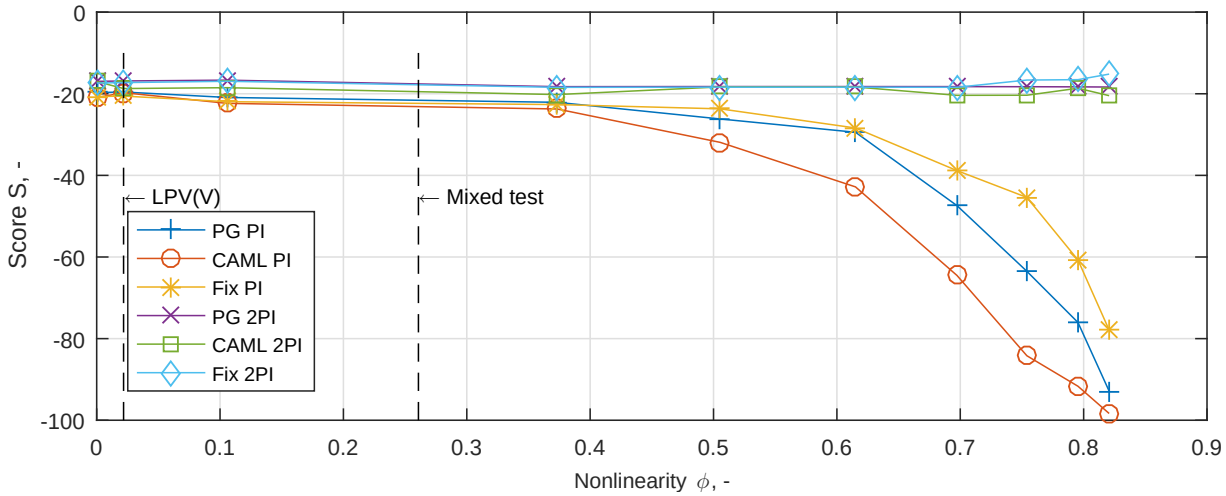


Figure 16: Sensitivity of the algorithms to nonlinearity (defined in the appendix). The PG algorithm is more robust to nonlinearity, with the double loop variant performing better.

system closer to its trimmed point. This limits the effect of the nonlinearity.

It is noticed that with a fixed cascaded PI controller, the score increases for high nonlinearity. This is an artifact of the specific nonlinearity that is introduced. If the linearized systems are compared to the linearized systems at a 0.1 m/s airspeed, it is found that the higher airspeed typically increases stability. Likewise, a lower airspeed often results in instability: if a -0.1 m/s step is taken, the nonlinear systems become more difficult to control.

V.E. Flapping

When a flapping motion is introduced, the performance of all algorithms suffers, as shown in figure 17. This figure also shows the performance during a simulation where the filter is installed, but flapping is not added. The flapping can be considered a disturbance, which cannot be filtered away completely. This affects the low-level PI controllers, as well as the learning processes. The time-varying model of the PG algorithm will be altered, which has an effect on the gradient. CAML suffers by identifying an incorrect model, leading to an incorrect gain selection. The effect of flapping on CAML seems to be significantly stronger. The PG algorithm has a tracking accuracy comparable to a fixed PI controller.

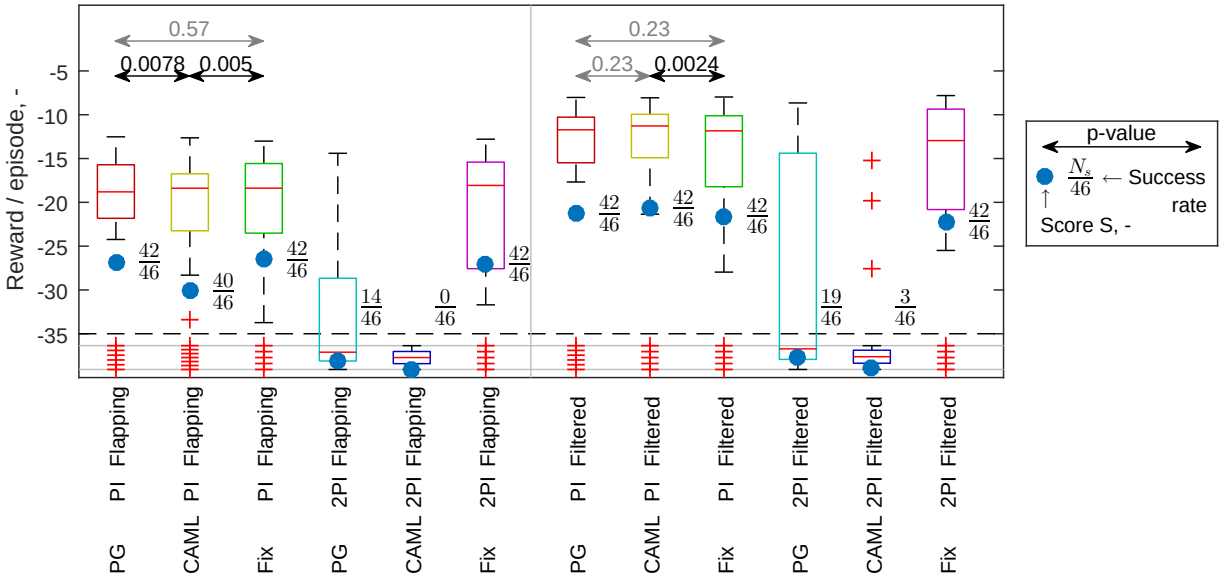


Figure 17: Results of the flapping test, compared to a test where the states are filtered, but without flapping. The statistical significance is shown for some differences in tracking accuracy. CAML performs worse than the other algorithms if flapping is present. Cascaded intelligent PI controllers perform worse with flapping.

For both algorithms, the performance of the cascaded controllers is degraded severely compared to the non-flapping simulations, which seems to conflict with the high performance of the fixed cascaded PI controller. A closer look at the individual fixed PI controllers in figure 18 provides an explanation. For the single loop PI controllers in figure 18a, the performance of each controller degrades to a comparable extent. The cascaded PI controllers in figure 18b, however, show a varying level of degradation. Most of the gain sets show large deterioration of performance, but some are robust. Because the algorithms do not account for flapping directly, they cannot anticipate this, resulting in incorrect gain selections. This is especially true for CAML.

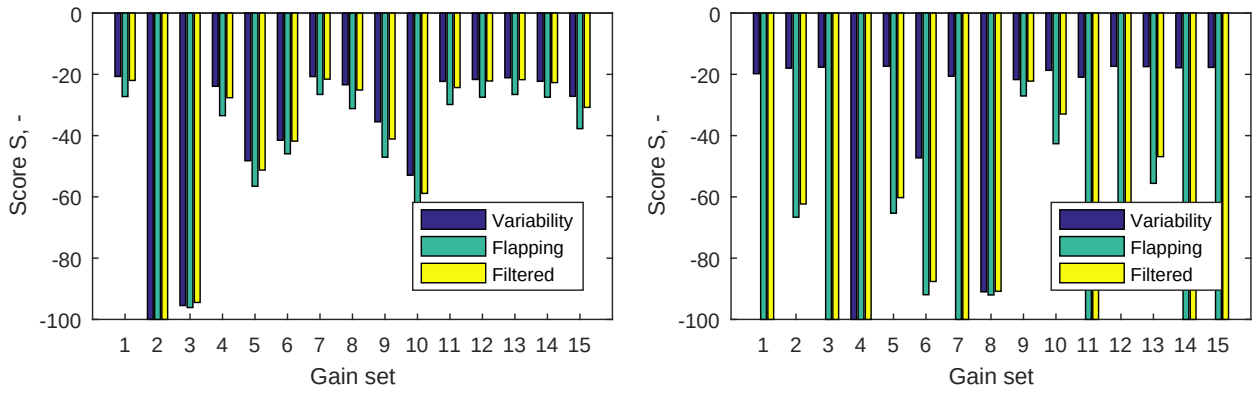
The fact that the performance with filter, but without flapping is also low, proves that this is due to the filtering process. The cascaded PI controllers, operating at higher frequencies, are more sensitive to this. Figure 19 shows the Bode diagram of the inner open loop (pitch angle control) with gain sets 1 or 9. Because the phase margin with gain set 1 is very small, it will become unstable if a phase lag is introduced by the filter.

V.F. Mixed test

In the final test, the noise, nonlinearity, and flapping are considered at the same time. Note that this test does not consider 46 different models, but only one model at 46 trim points. The results shown in figure 20 are therefore not directly comparable to the previous simulations, which also accounted for variability.

The single loop PG algorithm performs significantly better than CAML on the mixed test. CAML is affected more by both nonlinearity and flapping, resulting in lower performance. The difference between the PG algorithm and a fixed PI controller is not significant.

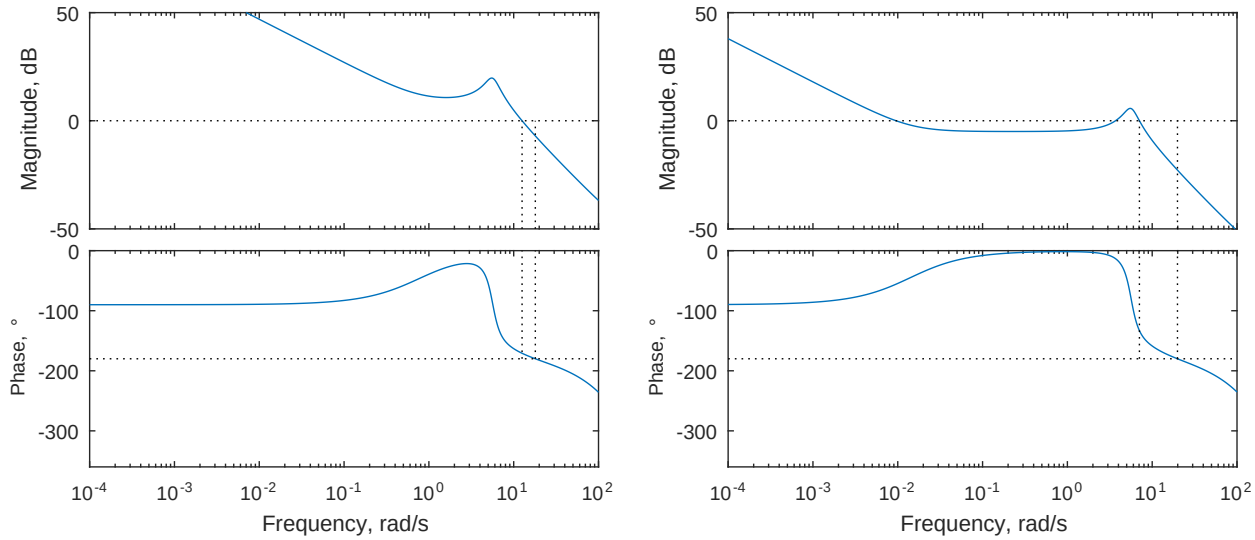
The cascaded versions of the algorithms are both showing lower performance. This is consistent with the findings of the flapping tests. It is therefore expected that this is caused by the filtering effect. It is again found that this effect is larger for CAML.



(a) Single loop PI

(b) Cascaded PI

Figure 18: Performance of the individual fixed PI controllers on the variability and flapping tests, and with filter only. Some of the cascaded PI controllers are very sensitive to the filtering, but the flapping itself has little influence.



(a) Gain set 1 (sensitive)

(b) Gain set 9 (insensitive)

Figure 19: Bode diagrams of the inner loop of an example DelFly with filter-sensitive gain set 1 and insensitive gain set 9. The small phase margin with gain set 1 becomes problematic if the filter is added.

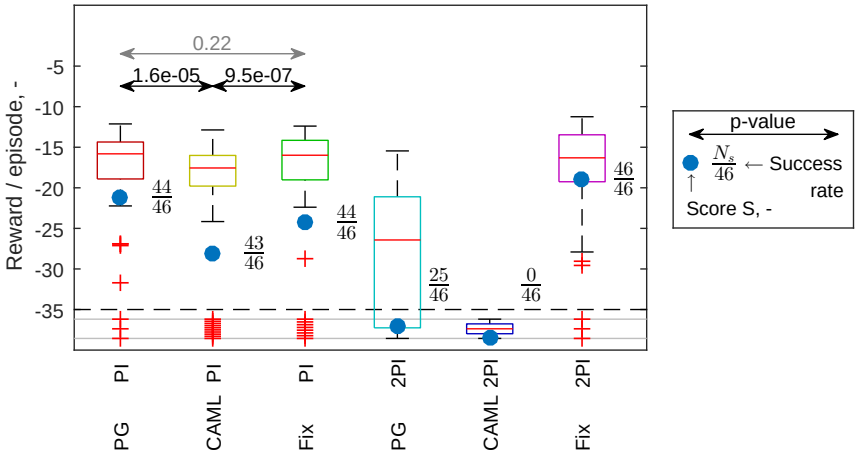


Figure 20: Results of the mixed test, where only one model is considered. CAML performs worse than the other algorithms.

VI. Conclusion and Recommendations

The DelFly is a difficult vehicle to control because of its flapping dynamics and nonlinearity. Also, there is variability between different individual vehicles, measurements are corrupted by noise, and disturbances change the state. Two Machine Learning control algorithms were used to tackle these challenges. First, a Policy Gradient algorithm of Reinforcement Learning is used, where the gradient is based on an inaccurate model. This approach is extended with a new variable learning rate technique in order to improve safety.

Second, CAML is proposed as a new algorithm to tackle such problems. This approach identifies a model, which is used to select the most appropriate PI gain set with a NN classifier. The selectable gain sets are predefined. This algorithm can change the gains rapidly during flight, which may improve safety. It is shown that identification of a stochastic model is not necessary. The RML1 algorithm is used to lower the effect of state noise during model identification, but this only works without flapping motion.

A baseline test, with 46 DelFly models, a fixed noise level, and no flapping dynamics, shows that the Policy Gradient approach results in more accurate tracking than a fixed PI controller. The PG algorithm has a tracking accuracy comparable to CAML, but this algorithm results in a lower safety on the test problem. This is because there is a universal gain set, which is able to control 42 out of 46 models. The remaining four models cannot be controlled by PI controllers of the selected structure, which means that CAML is also unable to control these, and cannot obtain a safety increase by its rapid gain changes. The PG algorithm is recommended for a problem with a universal gain set. It is recommended to test CAML on a problem without a universal gain set, and to ensure that all models can be controlled by at least one of the gain sets.

The PG algorithm has a similar robustness to noise as the predefined controller. CAML is more sensitive to noise, because this complicates model identification. Single loop CAML is more sensitive to disturbances than the other algorithms, and shows a fast decrease in performance. Single loop CAML is therefore not recommended for problems with high disturbance levels. The other algorithms perform comparably.

When flapping is introduced, single loop CAML performs slightly worse than a fixed PI controller, while the PG algorithm performs obtains a performance similar to a fixed controller. A low-pass filter is required, such that control occurs only at lower frequencies. This results in problems with the cascaded ML algorithms, because many combinations of models and gain sets have a low phase margin and become unstable due to the phase lag of the filter. It is recommended to test other types of filters, in order to improve real-time control. The Policy Gradient approach could be improved by explicitly using the filter to select the gradient direction. If unfiltered states are sampled, and the filter is applied afterwards, one can find a gradient direction accounting for filtering. CAML can be improved by accounting for filtering in the gain set generation, such that the selection takes into account filtering effects.

On a mixed test involving flapping, nonlinearity and noise, it is found that the PG algorithm performs as well as a fixed PI controller. CAML cannot match the performance of a fixed PI controller. Due to the filter, cascaded ML controllers show low performance. The filtering method should be improved for effective use of cascaded controllers.

The CAML algorithm may be improved by letting the experienced reward influence selection. It was found that model identification is difficult in the presence of filtering. It is possible to improve the model after each episode, by storing the states and applying the zero-phase filter offline. The selection process itself may be improved by testing other function approximators, e.g., deep NNs. The computational load may be alleviated by evaluating the classifier at a lower frequency than the low-level PI controller. A thorough analysis of computational time is recommended for future research.

A new algorithm that follows naturally from this research combines the CAML and PG algorithms sequentially. One may imagine a controller that uses CAML for the first instances, until the gain set remains constant for some time. Then, the gains can be fine-tuned using the PG algorithm. Such a method may improve both safety and tracking accuracy. Research into such a method is therefore recommended.

This research was limited to single or double loop PI controllers. However, literature has demonstrated the use of pitch (rate) dampers[3] and dynamic inversion-based concepts.[5] It should be investigated whether the ML algorithms can be implemented for such controllers as well. Also, this paper has only treated elevator control. New models are required to investigate the control of the flapping frequency and rudder.

Proving the effectiveness of the algorithms on simulation models is limited by the validity of the models of the dynamics, noise and disturbances. A final controller should be subjected to a physical flight test for validation. This paper is a step towards such a flight test. Promising results were obtained with the Policy Gradient algorithm. It is suggested to use this algorithm, with the newly developed variable learning rate technique, for further development of a DelFly controller.

Appendix

This section demonstrates how the nonlinearity metric is defined. An artificial nonlinearity is superposed on the linear system. This nonlinearity is based on the Linear Parameter-Varying (LPV) system in Eq. 15.

$$x(t+1) = (A_0 + A_v V)x(t) + (B_0 + B_v V)\delta_e(t) \tag{15}$$

The parameters of the A- and B-matrices are assumed to be linearly dependent on the airspeed. Although the nonlinearity is artificial, it is based on the 46 models, by examining the variation of the aerodynamic parameters with the trimmed airspeeds of the models. Figure 21 shows the variation of three aerodynamic parameters with airspeed.

The LPV model can be estimated by performing linear regression on the data in figure 21. This is performed with a univariate linear function for all aerodynamic parameters, resulting in the A_0-, A_v-, B_0- and B_v-matrices. These matrices form an unvalidated LPV model of the DelFly. Validation of this model is not the focus of this work. However, the A_v- and B_v matrices provide an indication of the level of nonlinearity of the model. A scaled version of the nonlinear part of Eq. 15 is superposed on all 46 linear models, as shown in Eq. 16.

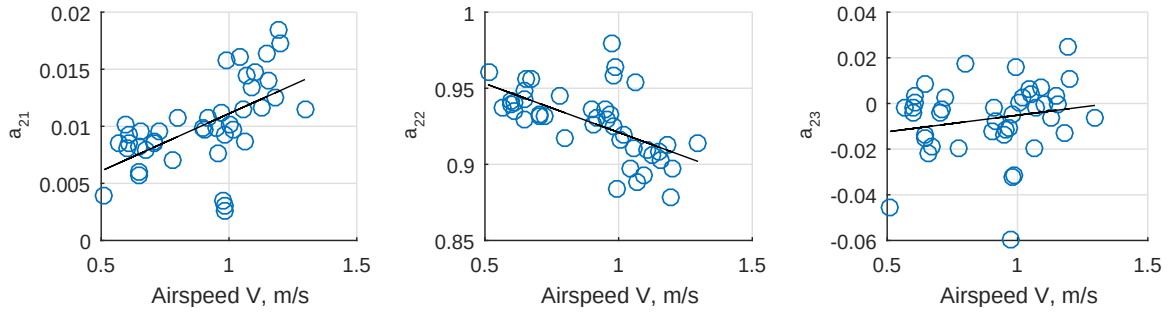
$$x_i(t+1) = (A_i + \Lambda A_v(V - V_{0,i}))x(t+1) + (B_i + \Lambda B_v(V - V_{0,i}))\delta_e(t) \tag{16}$$

By varying the scaling parameter Λ, and repeatedly performing the tracking analysis, the sensitivities of the algorithms to nonlinearity can be analyzed. With $\Lambda = 0$, the 46 local linear models are obtained, while $\Lambda = 1$ corresponds to the expected level of nonlinearity.

While Λ provides an indication of the level of nonlinearity, it is only applicable to models of the form of 15. A more general metric is required in order to allow the use of other model structures. A generic metric[25] is based on the difference between the linear system and the nonlinear system. It requires the definition of a nonlinear system N, with $y_N = N\left[u, x_{N,0}\right]$, and a linear system G, with $y_G = G\left[u, x_{G,0}\right]$. The y and u are signals, with a time dimension. The operators N and G map the scalar input signal u to the vector output signal y, starting from initial conditions $x_{N,0}$ and $x_{G,0}$. The spaces $\mathcal{U}_a$ and $\mathcal{Y}_a$ are defined as the allowable spaces of the signals u and y. Furthermore, the spaces $\mathcal{X}_{0,a}$ and $\mathcal{X}_{0,G}$ are the allowable spaces of the initial states of the nonlinear and linear systems, respectively. Finally, $\mathcal{G}$ is the space of allowable linear systems. The nonlinearity measure used in this research[25] is presented in Eq. 17.

$$\begin{aligned} \phi(t_f) = \inf_{G\in\mathcal{G}} \sup_{u,x_{N,0}\in\mathcal{S}} \inf_{x_{G,0}\in\mathcal{X}_{0,G}} \frac{\left\|G\left[u, x_{G,0}\right] - N\left[u, x_{N,0}\right]\right\|_z}{\left\|N\left[u, x_{N,0}\right]\right\|_z} \\ \text{with } \mathcal{S} = \{(u, x_{N,0}) :\ u \in \mathcal{U}_a,\ x_{N,0} \in \mathcal{X}_{0,a},\ N\left[u, X_{N,0}\right] \in \mathcal{Y}_a\} \end{aligned} \tag{17}$$

The measure ϕ takes a suitable norm of the difference signal between N and G, and normalizes it by the norm of the nonlinear output signal. This is a complex optimization problem, because it requires



(a) a_{21} (b) a_{22} (c) a_{23}

Figure 21: Generation of the artificial nonlinearity for three aerodynamic parameters. Aerodynamic parameters of the 46 models are shown, together with the estimated trend.

optimization over all allowable linear models, with all allowable initial linear states. This is done for the worst-case scenario of the input signal and initial nonlinear state, and requires the norm of a signal from $t = 0$ to $t = t_f$. Directly calculating the solution of this problem is practically infeasible.[25]

In this research, the problem is solved by limiting the allowable spaces of the signals and states to a single point. For each of the 46 models, there is a trimmed state $x_{0,i}$. The limitation is enforced that $x_{N,0} = x_{G,0} = x_{0,i}$ for each of the models. Furthermore, this paper only considers one allowable input signal u_s, which is a 0.1 rad elevator step. If the responses of N and G to this step are similar, it is expected that the models will be similar in general. Finally, the 'best' linear model is fixed; the 46 linear models F provide a reasonable approximation of the optimal linear model[j]. The final time is set to $t_f = 1$ s. This removes all infimum and supremum operators, arriving at Eq. 18 for the nonlinearity metric used in this research.

$$\phi_i = \frac{\|F\left[u_s, x_{0,i}\right] - N\left[u_s, x_{0,i}\right]\|_z}{\|N\left[u_s, x_{0,i}\right]\|_z} \tag{18}$$

The norm that is used represents the RMS values of the signals, where each state is scaled by the reference RMS value used for the noise metric. This norm is presented in Eq. 19.

$$\|y\|_z = \sqrt{\sum_{t=0}^{t_f} \sum_{i=1}^{4} \left(\frac{y_i}{y_{i,ref}}\right)^2} \tag{19}$$

The metric ϕ_i is calculated for each of the 46 models. For each Λ, the median of the 46 ϕ_i is used as the nonlinearity.

Acknowledgments

The authors would like to thank Frank Rijks for developing and demonstrating the DelFly in figure 1.

References

[1]De Croon, G. C. H. E., Perçin, M., Remes, B. D. W., Ruijsink, R., and De Wagter, C., *The DelFly: Design, Aerodynamics and Artificial Intelligence of a Flapping Wing Robot*, Springer Netherlands, 1st ed., 2016.

[2]De Croon, G. C. H. E., Groen, M. A., De Wagter, C., Remes, B. D. W., Ruijsink, R., and Van Oudheusden, B. W., "Design, Aerodynamics and Autonomy of the DelFly," *Bioinspiration & Biomimetics*, Vol. 7, No. 2, May 2012.

[3]De Wagter, C., Koopmans, J. A., De Croon, G. C. H. E., Remes, B. D. W., and Ruijsink, R., "Autonomous Wind Tunnel Free-Flight of a Flapping Wing MAV," *Advances in Aerospace Guidance, Navigation and Control: Selected Papers of the Second CEAS Specialist Conference on Guidance, Navigation and Control*, edited by Q. P. Chu, J. A. Mulder, D. Choukroun, E. van Kampen, C. C. de Visser, and G. Looye, April 2013, pp. 603–621.

[4]Verboom, J. L., Tijmons, S., De Wagter, C., Remes, B. D. W., Babuška, R., and De Croon, G. C. H. E., "Attitude and Altitude Estimation and Control on board a Flapping Wing Micro Air Vehicle," *IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 5846–5851.

[5]Cunis, T., Karásek, M., and De Croon, G. C. H. E., "Precision Position Control of the DelFly II Flapping-wing Micro Air Vehicle in a Wind-tunnel," *International Micro Air Vehicle Conference and Competition (IMAV)*, Oct. 2016.

[6]Junell, J. L., Mannucci, T., Zhou, Y., and Van Kampen, E., "Self-tuning Gains of a Quadrotor using a Simple Model for Policy Gradient Reinforcement Learning," *AIAA Guidance, Navigation, and Control Conference*, Jan. 2016.

[7]Caetano, J. V., De Visser, C. C., De Croon, G. C. H. E., Remes, B. D. W., De Wagter, C., Verboom, J., and Mulder, M., "Linear Aerodynamic Model Identification of a Flapping Wing MAV based on Flight Test Data," *International Journal of Micro Air Vehicles*, Vol. 5, No. 4, Dec. 2013, pp. 273–286.

[8]Armanini, S. F., De Visser, C. C., De Croon, G. C. H. E., and Mulder, M., "Time-Varying Model Identification of Flapping-Wing Vehicle Dynamics Using Flight Data," *Journal of Guidance, Control, and Dynamics*, Vol. 39, No. 3, March 2016, pp. 526–541.

[9]Chang, J., Armanini, S. F., and De Visser, C. C., "Feasibility of LTI-Model-Based LPV Model of DelFly," Unpublished MSc paper.

[10]Van Kampen, E., Chu, Q. P., and Mulder, J. A., "Continuous Adaptive Critic Flight Control Aided with Approximated Plant Dynamics," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Aug. 2006.

[11]Sutton, R. S. and Barto, A. G., *Reinforcement Learning: An Introduction*, MIT press, 1st ed., 1998.

[j]This approximation is not perfect, because the step input causes the state to vary. The real optimal linear model would have a different trimmed condition, but is difficult to determine.

[12]Motamed, M. and Yan, J., "A Reinforcement Learning Approach to Lift Generation in Flapping MAVs: Experimental Results," *IEEE International Conference on Robotics and Automation (ICRA)*, April 2007, pp. 748–754.

[13]Motamed, M. and Yan, J., "A Reinforcement Learning Approach to Lift Generation in Flapping MAVs: Simulation Results," *IEEE International Conference on Robotics and Automation (ICRA)*, May 2006, pp. 2150–2155.

[14]Roberts, J. W., Moret, L., Zhang, J., and Tedrake, R., "Motor Learning at Intermediate Reynolds Number: Experiments with Policy Gradient on the Flapping Flight of a Rigid Wing," *From Motor Learning to Interaction Learning in Robots*, edited by O. Sigaud and J. Peters, Vol. 264 of *Studies in Computational Intelligence*, Springer Berlin Heidelberg, 2010, pp. 293–309.

[15]Wang, J. and Kim, J., "Optimization of Fish-like Locomotion using Hierarchical Reinforcement Learning," *International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Oct. 2015.

[16]Abbeel, P., Quigley, M., and Ng, A. Y., "Using Inaccurate Models in Reinforcement Learning," *International Conference on Machine Learning (ICML)*, June 2006.

[17]Armanini, S. F., Karásek, M., De Visser, C. C., De Croon, G. C. H. E., and Mulder, M., "Flight Testing and Preliminary Analysis for Global System Identification of Ornithopter Dynamics using On-board and Off-board Data," *AIAA Atmospheric Flight Mechanics Conference*, Jan. 2017.

[18]Lupashin, S., Schöllig, A., Sherback, M., and D'Andrea, R., "A Simple Learning Strategy for High-speed Quadrocopter Multi-flips," *IEEE International Conference on Robotics and Automation (ICRA)*, May 2010, pp. 1642–1648.

[19]Koppejan, R. and Whiteson, S., "Neuroevolutionary Reinforcement Learning for Generalized Control of Simulated Helicopters," *Evolutionary Intelligence*, Vol. 4, No. 4, Dec. 2011, pp. 219–241.

[20]Moldovan, T. M., Levine, S., Jordan, M. I., and Abbeel, P., "Optimism-driven Exploration for Nonlinear Systems," *IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 3239–3246.

[21]Arthur, D. and Vassilvitskii, S., "k-means++: The Advantages of Careful Seeding," *ACM-SIAM Symposium on Discrete Algorithms*, Jan. 2007.

[22]Gustavsson, I., Ljung, L., and Söderström, T., "A Comparative Study of Recursive Identification Methods," Tech. Rep. TFRT; Vol 3085, Department of Automatic Control, Lund Institute of Technology (LTH), Jan. 1974.

[23]Friedlander, B., "System Identification Techniques for Adaptive Signal Processing," *Circuits, Systems and Signal Processing*, Vol. 1, No. 1, 1982, pp. 3–41.

[24]Wilcoxon, F., "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, Vol. 1, No. 6, Dec. 1945, pp. 80–83.

[25]Helbig, A., Marquardt, W., and Allgöwer, F., "Nonlinearity Measures: Definition, Computation and Applications," *Journal of Process Control*, Vol. 10, No. 2, April 2000, pp. 113–123.

4

Reflection on classification control

This chapter describes some issues regarding the CAML algorithm. Section 4.1 compares the two model identification algorithms considered in Chapter 3. The models obtained during the course of the identification are analyzed in Section 4.2. Section 4.3 contains an extensive analysis into the question whether the NNs are overfitting. The behavior of the filter is the subject of Section 4.4.

4.1. Model identification algorithms

The Recursive Least Squares algorithm allows online identification of a dynamic model. This research considers discrete models, that predict the next state from the current state and control input. A major limitation of RLS is the inherent assumption that the current state is noise-free. This cannot be satisfied perfectly during system identification, which introduces inaccuracies into the model. It was found that these model imperfections cause problems with CAML.

The variance-based version of CAML can deal with imperfect models, as long as the imperfection can be estimated. The parameter covariance matrix in the RLS algorithm can be used for this. Eq. 4.1 defines σ_m^2 as a measure of the total variance of the model, namely the sum of all parameter variances. Every parameter variance is the expectation of the squared deviation of the corresponding parameter from its true value. Therefore, σ_m^2 is the expectation of the imperfection measure ϵ^2 defined in Eq. 4.2.

$$\sigma_m^2 = \sum_{i=1}^{4}\sum_{j=1}^{4}\sigma_{a_{ij}}^2 + \sum_{k=1}^{4}\sigma_{b_k}^2 \tag{4.1}$$

$$\epsilon^2 = \sum_{i=1}^{4}\sum_{j=1}^{4}\left(\hat{a}_{ij} - a_{ij}\right)^2 + \sum_{k=1}^{4}\left(\hat{b}_k - b_k\right)^2 \tag{4.2}$$

With σ_m^2 being the expectation of ϵ^2, one would expect these measures to have the same order of magnitude. Figure 4.1 shows the measures when RLS is used on a (partial) episode of a CAML controller on an example DelFly model. It can be seen that the model imperfection keeps increasing, but the parameter variance does not. This effect occurs on nearly all 46 DelFly models. The effect disappears if noise is removed. The model is thought of as a mapping $x(t), u(t) \rightarrow x(t+1)$. It is found that the increase in model imperfection is linked to the noise on $x(t)$, but not to the noise on $x(t+1)$. It is therefore due to the assumptions of the RLS estimator.

RML1 is implemented as an alternative. If the same simulation is performed with RML1, Figure 4.2 is obtained. The figure shows that RML1 effectively lowers the model imperfection compared to the simulation with RLS. This is found for a wide range of DelFly models. Therefore, RML1 is adopted as the identification algorithm in CAML.

When flapping is introduced, a low-pass filter must be added to isolate the time-averaged model. It is straightforward to implement this in the RLS algorithm, by using filtered versions of $x(t)$, $u(t)$ and $x(t+1)$ for model identification. For RML1, this is non-trivial, because a model-predicted $\hat{x}(t)$ is used. If $\hat{x}(t)$ is not filtered, it is not affected by the phase shift of the filter, while $u(t)$ and $x(t+1)$ are. However, $\hat{x}(t)$ is obtained from the time-averaged model, and therefore contains little high-frequency content. Filtering it again would

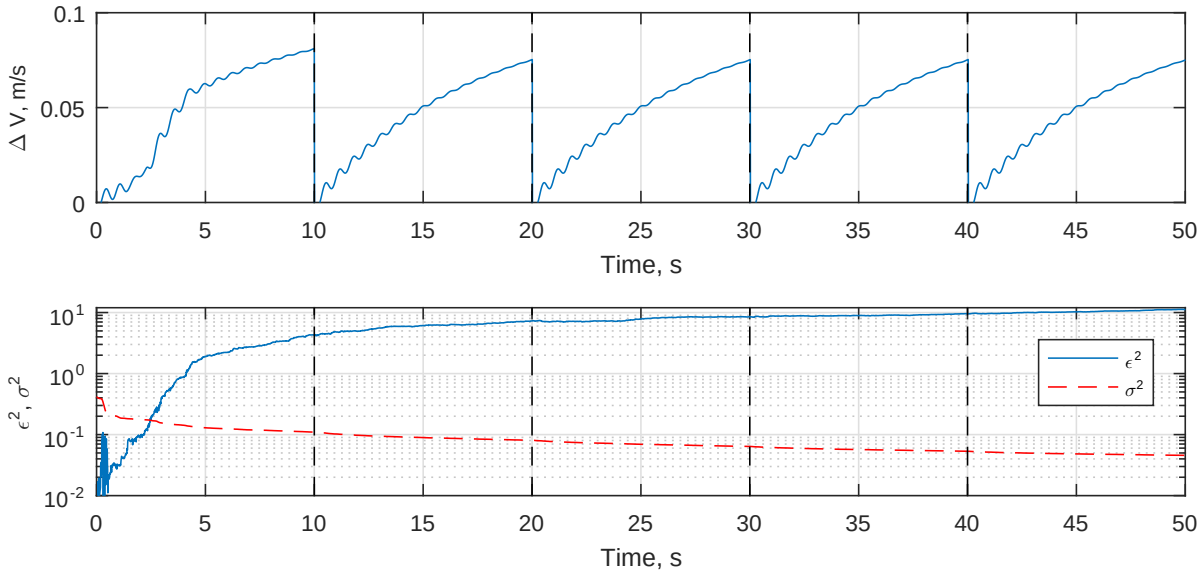


Figure 4.1: Model identification with RLS during a CAML test run without flapping.

also affect model identification. Another difficult matter is whether the filtered or unfiltered $u(t)$ should be used for prediction in the model.

After trying many combinations, it was found that the best results are obtained if a filtered version of $\hat{x}(t)$ is used for identification, and an unfiltered $u(t)$ for prediction of $\hat{x}(t+1)$. Identification with filtered RLS in Figure 4.3 is compared to identification with filtered RML1 in Figure 4.4. It can be seen that now, RLS obtains better results, but both methods result in problems. This issue could be solved with a better filtering technique. In this research, RLS is used for the simulations that include flapping.

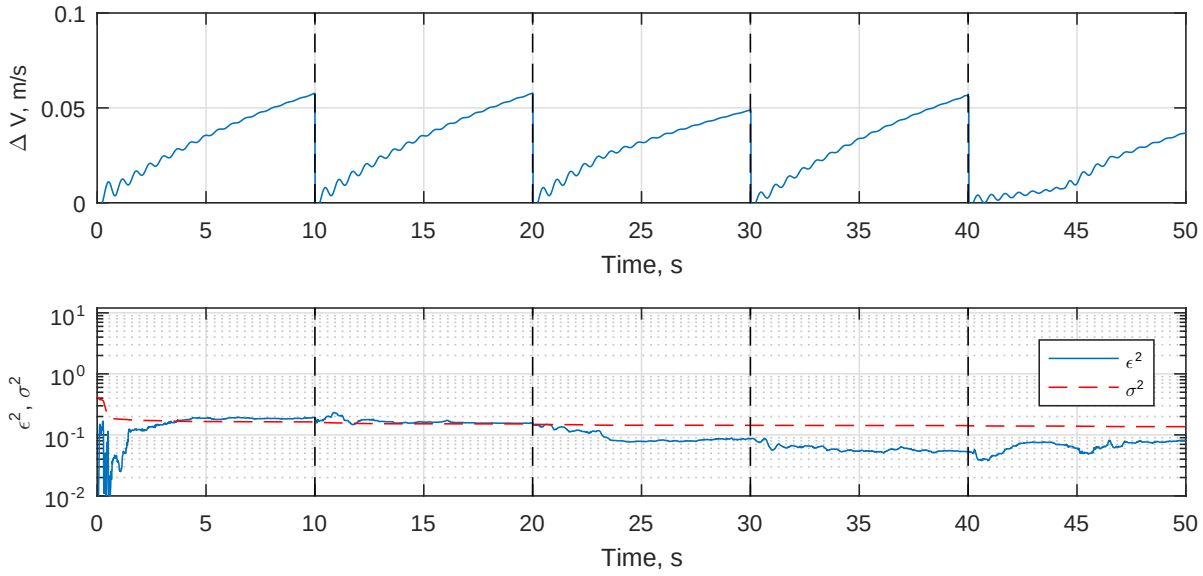


Figure 4.2: Model identification with RML1 during a CAML test run without flapping.

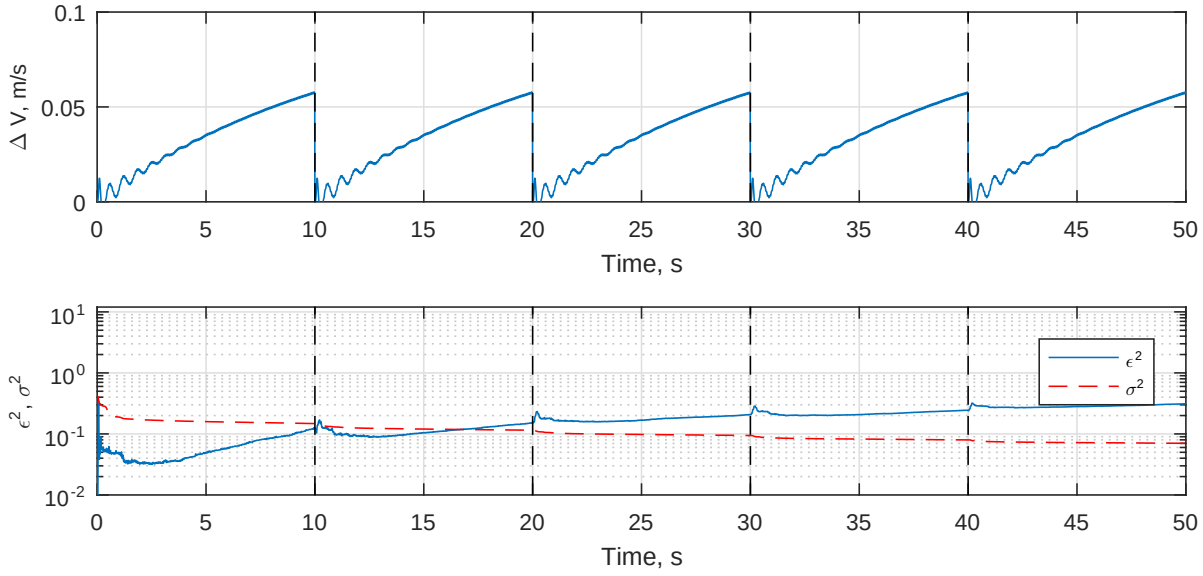


Figure 4.3: Model identification with RLS during a CAML test run with flapping. Filtered state is shown.

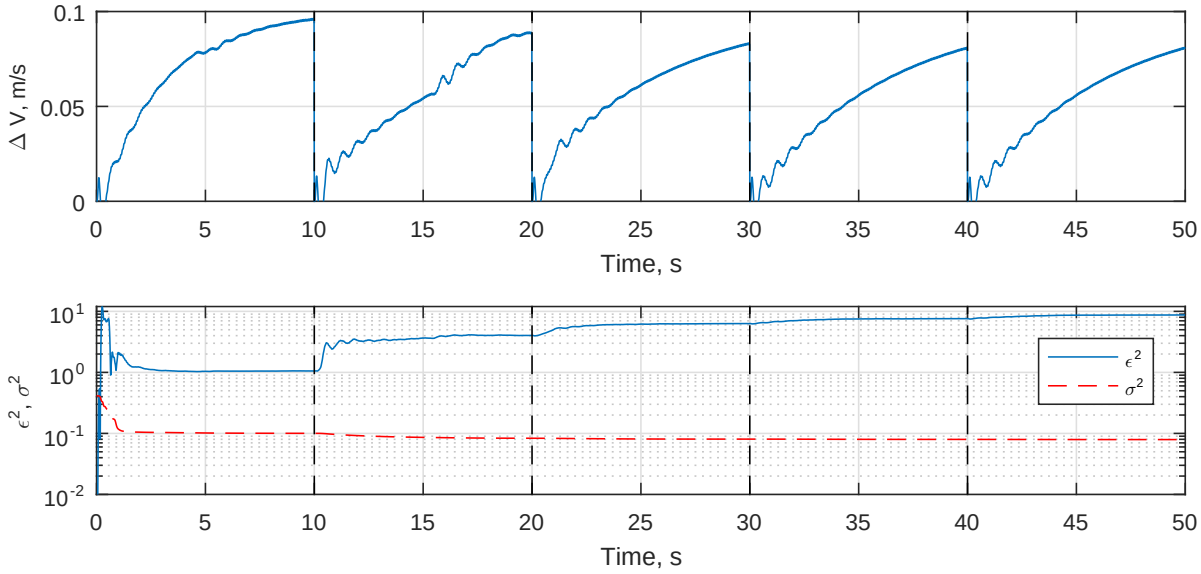


Figure 4.4: Model identification with RML1 during a CAML test run with flapping. Filtered state is shown.

4.2. Intermediate models

The principle of CAML is that the selection of the gains improves over time, due to improving model identification. The model is initialized at the average DelFly system. If the estimator converges, the model will become close to the real system. However, it is uncertain what happens in between. There will be many 'intermediate models'. If these intermediate models do not represent anything physical, which is close to the DelFly, CAML will select very inappropriate gains during the intermediate stage.

In the variance-based version of CAML, suitable training pairs of models and variances are determined by initializing the model at one system, and letting it converge to another system. This is also a reason to analyze the intermediate models. It is important that the intermediate identified models represent something close to the DelFly. Otherwise, the Neural Network is trained for non-physical systems.

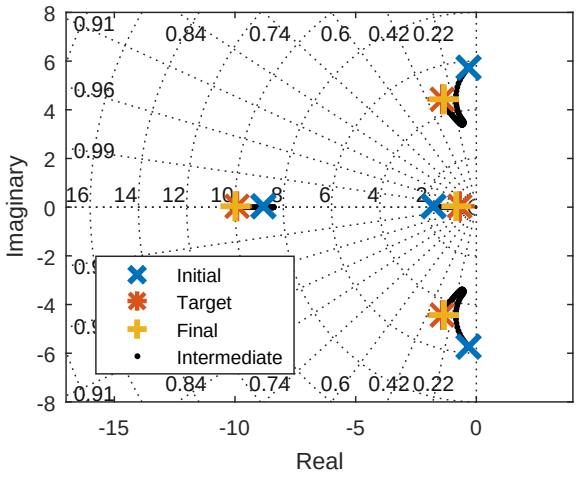
In order to analyze whether the intermediate models are physical, the poles of the A-matrices are plotted. The models are first converted to continuous time for illustrative purposes. Figure 4.5 shows how the poles of the models move during identification. This can be compared to Figure 4.6, which shows the poles of all models. These models are identified with both Ordinary Least Squares (OLS) and Maximum Likelihood Estimation (MLE).

Figure 4.5a shows the intermediate models during identification with RLS, if noise is removed. Note that this is just one example; with different initializations and target systems, different but similar behavior is found. The real poles move over the real axis towards the targets[1]. The complex poles also move towards the target, but do not move in a straight line. The 'loop' found in between does not have a physical meaning, and is different for every combination of systems. Nevertheless, the poles stay reasonably close to the initial positions and targets, and it is concluded that the intermediate models are physical. Figure 4.5b shows that with RML1, stranger detours are taken.

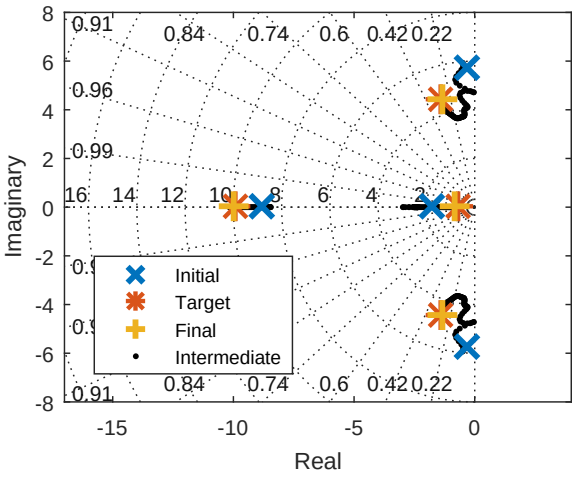
If noise is added, Figure 4.5c is obtained with RLS. Now, the poles do not finish on the target positions (even though they are quite close in this example). Noise results in a wider spread of the intermediate models. Especially the real poles vary a lot in this case, and even become unstable. The complex poles take a quick jump, and then follow a curved path towards the target. The unstable intermediate models can be considered non-physical, because both the initial and target system are stable. However, this is only during the very first instants of identification. The simulation results show that CAML is usually able of surviving this stage without crashing the DelFly.

When RML1 is used on a noisy dataset, Figure 4.5d is obtained. As in the noise-free case, RML1 results in a wider spread of intermediate models than RLS. The real poles become oscillatory for some time, but eventually move towards their targets. For the complex poles, no clear path can be recognized. Those poles become unstable at some stages, and highly damped at others. Nevertheless, the intermediate poles are mostly centered around the target locations. Because the poles remain reasonably close to the physical locations, it is expected that the identification process is good enough for effective use of CAML. The simulation runs confirm this.

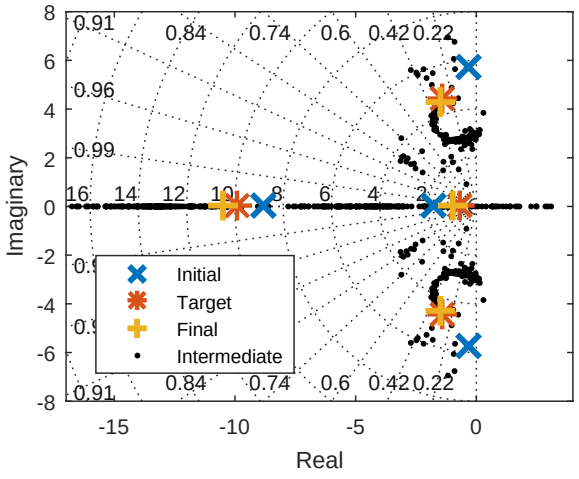
[1] Different runs show that sometimes, the intermediate poles are complex.



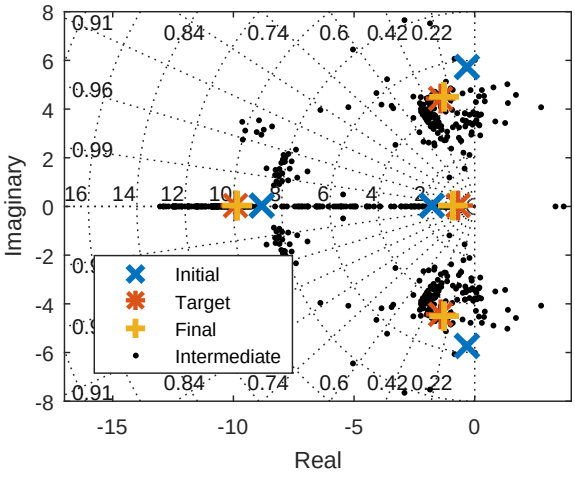
(a) RLS, without noise.



(b) RML1, without noise.



(c) RLS, with noise.



(d) RML1, with noise.

Figure 4.5: Motion of the continuous-time poles of the identified model during model identification.

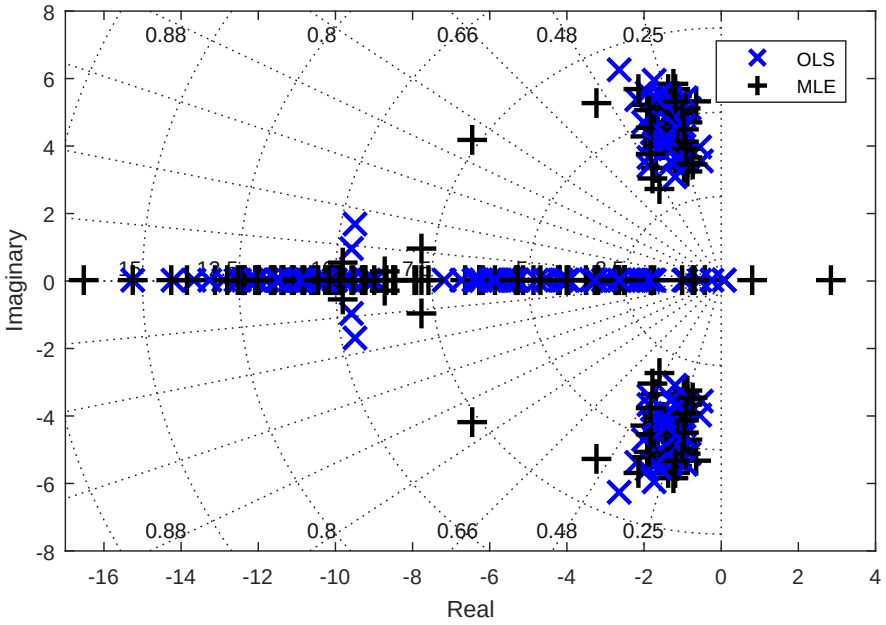


Figure 4.6: Poles of all 46 models, for both Ordinary Least Squares and Maximum Likelihood Estimation.

4.3. Neural network overfitting

It is essential for the behavior of the CAML controller that the classification Neural Network is sufficiently accurate. It is trained on a dataset of a given size. Problems will be found if overfitting occurs. In that case, the NN will work well on the training dataset, but will not work well on new datasets. Overfitting occurs when the complexity reachable by the classifier is higher than the complexity in the dataset, and the size of the dataset is too small.

For CAML, this means that the number of neurons and the dataset size must be considered together. The most effective way to avoid overfitting is to increase the size of the dataset. The dataset is randomly generated in this research, and there is no absolute limit on its size. However, computational time forms a practical limit. For a given size of the dataset, multiple tools are available against overfitting.

In this research, the main tool to avoid overfitting is early stopping. This means that the total dataset available for training is split up into a training set and a validation set. The training procedure (scaled conjugate gradient) requires gradients to be found. They are calculated for the training dataset only. If the step size is small enough, this will lead to an improved performance on the training dataset. The principle of early stopping is that the performance on the validation dataset is also checked. Training stops when the parameter updates (based on the training set) do no longer lead to improved performance on the validation set. This research uses 80% of the data for training, and 20% for validation.

Early stopping may fail when there are random regularities affecting both the training and validation set. The number of neurons must still be selected appropriately, such that new datapoints outside both datasets will be handled well. Two tests are performed to assess the correct number of neurons.

First, a second dataset of 10000 models is generated. This is the test dataset, which is independent of the training and validation set. Neural networks of various sizes are trained on the first dataset with early stopping. This is done ten times, where the first dataset is randomly divided into a training set and validation set each time. Next, they are tested on the complete second dataset. The performance is shown in Figure 4.7. This figure was already shown in Chapter 3, but is analyzed in a broader context of overfitting in this section.

For the numbers of neurons considered, the error rate on the test set keeps decreasing steadily. There seems to be no indication that overfitting is problematic. This figure is also used for tuning the number of neurons. It is decided that 100 neurons is sufficient. The error rate is lower with 200 neurons, but this would increase the (online) computational time.

In the second test, early stopping is not used at all. Instead, the NNs are trained to a fixed performance level. The first dataset is used for training. A random 20% of this set is disregarded, such that the same number of models as before are available for training. The next step, as before, is to test the trained NNs on the second dataset. This test is required to check if the number of neurons is really appropriate for the complexity and size of the dataset, even if no early stopping is used.

The scaled conjugate gradient training procedure is used to minimize the cross-entropy. Therefore, the cross-entropy is the suitable performance measure for this test. During the first test, it is found that the validation stop usually occurs when the cross-entropy is around 0.05 for the PI controllers, and 0.035 for the cascaded PI controllers. During the second test, all NNs are trained to these performance values, without using early stopping. If this value has not been reached, training is halted after 1000 episodes. The results of

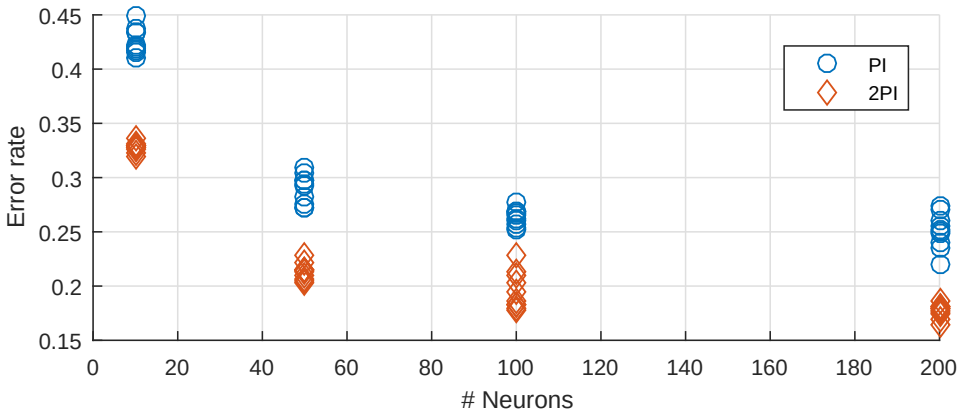


Figure 4.7: Tuning of the number of gain sets. Overfitting does not seem to be a problem. A number of 100 neurons is considered sufficient.

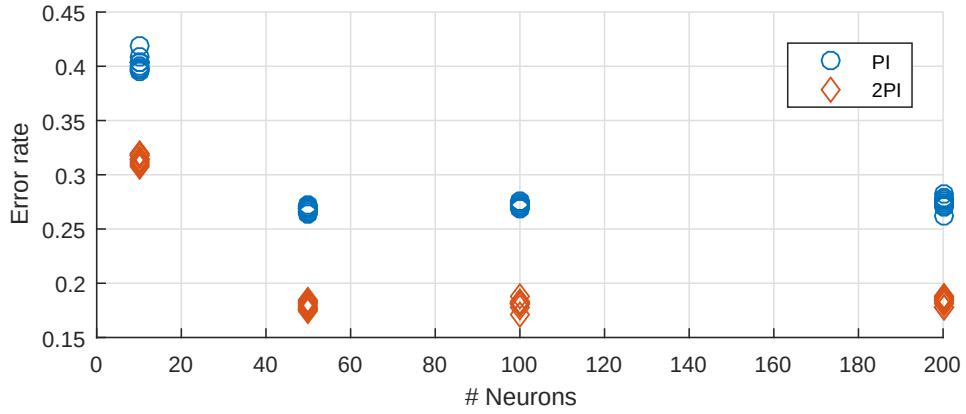


Figure 4.8: Performance on the test set if networks are trained to a constant cross-entropy on the training set.

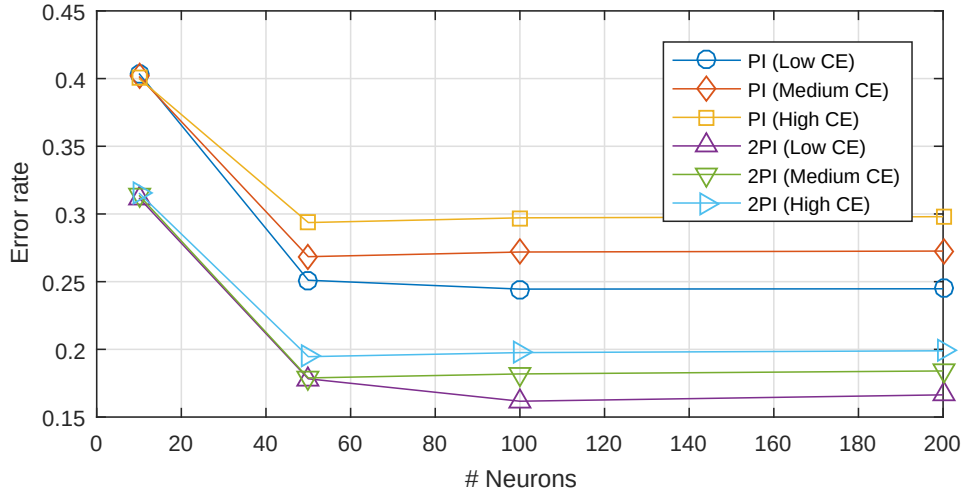


Figure 4.9: Performance on the test set if networks are trained to a constant cross-entropy on the training set, for different levels of final cross-entropy (CE). *Low* and *high* cross-entropy are 10% lower and higher than medium cross-entropy, respectively.

the cross-entropy test are shown in Figure 4.8.

It can be seen that the error rate steadily decreases for low numbers of neurons. At numbers of neurons higher than 100, the error rate stabilizes. Since it does not yet increase, there is no indication that overfitting is problematic with 200 neurons. The performance at 100 neurons is nearly equal. Therefore, a number of 100 neurons is the preferred value.

A weakness of this test is that the constant cross-entropy, to which all networks are trained, is a free parameter. Incorrect selection may affect the results. Different values have been tried to ensure that the test works correctly. Cross-entropy values that are 10% lower or higher are attempted. The results are shown in Figure 4.9. Only the median error rates for each setting are shown.

With 10 neurons, the error rates are the same for all Cross-Entropy (CE) settings. This is because the required performance can never be met, even if it is 10% higher; 10 neurons is simply not enough. For the other numbers, all cross-entropy settings result in the same behavior: a steady decrease, followed by stabilization after 100 neurons. It is therefore concluded that overfitting is not an issue when 100 neurons are used.

4.4. Filtering phase shift

It was mentioned in Chapter 3 that the filtering causes a phase shift in the signal. This section elaborates this claim with a figure.

In order to remove the flapping-related time-varying dynamics, a low-pass filter is required. For identification of the local linear models [3], this was done using a fourth order Butterworth filter, with a cut-off frequency of 6 Hz. Figure 4.10 shows some flight data, together with the filtered versions.

The identification was performed offline. It is therefore possible to use a non-causal filter. The zero-phase

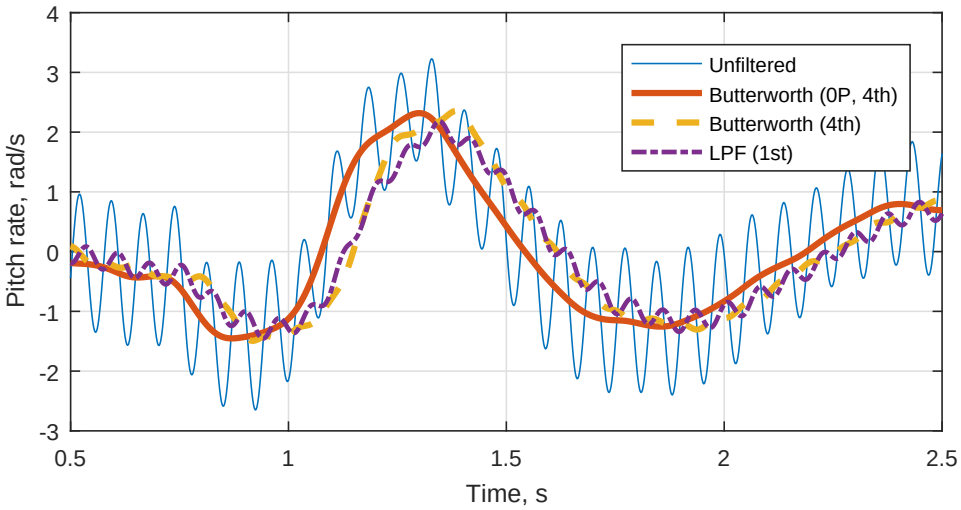


Figure 4.10: Different filters applied to the flight data. The zero-phase (0P) filter is clearly preferable.

filtering procedure used filters the signal both backward and forward in time. This results in a filtered signal without a phase shift.

If online filtering is required, a causal filter is needed. Figure 4.10 also shows how the causal variant of the same fourth order Butterworth filter performs. This signal still shows some flapping-related content, and more importantly, has a significant phase delay.

It was attempted to lower the order of the filter to remove this effect. However, even a standard discrete first order low-pass filter shows a significant delay. The delay is hardly reduced, but the flapping motion is much stronger. This will make identification even more difficult.

In order to apply online model identification effectively, the filtering procedure should be improved. It is possible to store the history of the states, and apply the zero-phase filter at every time step, but this is computationally prohibitive. Improving the filtering procedure is an important recommendation for future research involving CAML for flapping wing vehicles.

5

Assessment of recommendations

The scientific paper in Chapter 3 has led to several recommendations for the improvement of the algorithms. This chapter contains some preliminary analyses regarding these recommendations. Section 5.1 analyzes the effect of filtered training of CAML on the performance of the cascaded PI controllers, and Section 5.2 assesses a combination of CAML and the Policy Gradient algorithm.

5.1. Filtered classifier training

It was found in Chapter 3 that CAML performs poorly with cascaded PI controllers when the low-pass filter is installed. This seems to be due to the phase shift induced by the filter. However, it was also found that the fixed cascaded PI controllers do perform well in some cases. CAML may select better gain sets if it accounts for the filtering effect explicitly. This section contains a first analysis in this direction.

There are two components of CAML where the filtering effect can be introduced explicitly. First, the generation of suitable gain sets is altered. The gain sets are determined by performing offline gradient-based optimization on the 46 known models; simulations are performed for many trial gain settings. These simulations can easily be modified by including the low-pass filter in the feedback loops. This leads to more suitable gain sets to select from.

Second, the training of the Neural Network requires example selections, which are generated by performing simulations on 40000 test models. These simulations should also include the filter. In this way, the NN classifier can account for filtering in its selection.

With these two modification, the selection procedure fully accounts for the filtering effect. However, the model identification procedure is still affected by the filtering and flapping, and inadequate model identification may still lead to incorrect selection.

Figure 5.1 shows the performance of the modified CAML, compared to the other algorithms, if flapping and filtering are added. The LPF-trained CAML performs much better than the original CAML and the PG algorithm. However, it obtains a lower score than the fixed controller, because the success rate is lower. The difference in tracking accuracy is not significant.

The second part of Figure 5.1 describes the test where the filter is installed, but no flapping is present. Here, the modified CAML algorithm outperforms all other algorithms. Compared to the fixed controller, the success rate is equal, but the tracking accuracy is significantly higher. The success rate found is the maximum success rate that can be obtained with a double loop PI controller. It can therefore be concluded that the filtering modification to CAML effectively counteracts the filtering phase delay.

The performances of the fixed controller and the modified CAML algorithm can be compared between the tests with and without flapping motion. When flapping is added, both algorithms show a decrease in tracking accuracy. A straightforward explanation for this is the fact that the flapping motion causes a periodical variation around the intended motion, which results in a punishment. However, the modified CAML algorithm also shows a decrease in success rate, which is not found with the fixed controller. It is concluded from this that the flapping motion itself also results in improper gain selection. This is because the low-pass filter cannot remove all flapping-related motion from the signals [2]. The remaining periodical signal part affects the model identification, leading to incorrect selections.

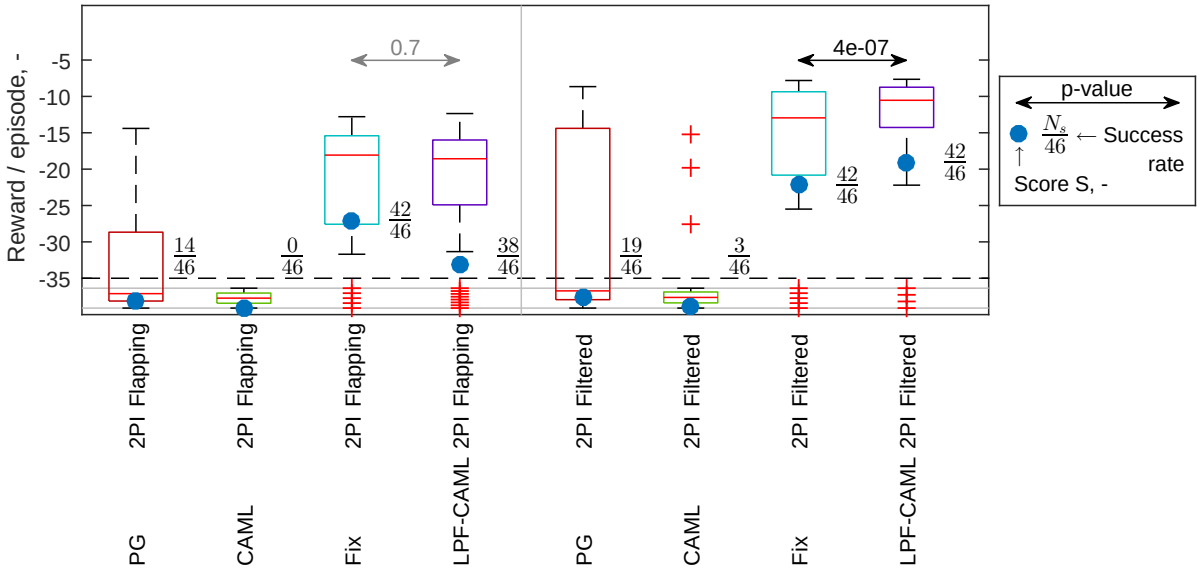


Figure 5.1: Performance of CAML trained with a Low-Pass Filter on the *Flapping* test. A test with the filter installed, but without flapping motion, is shown for comparison.

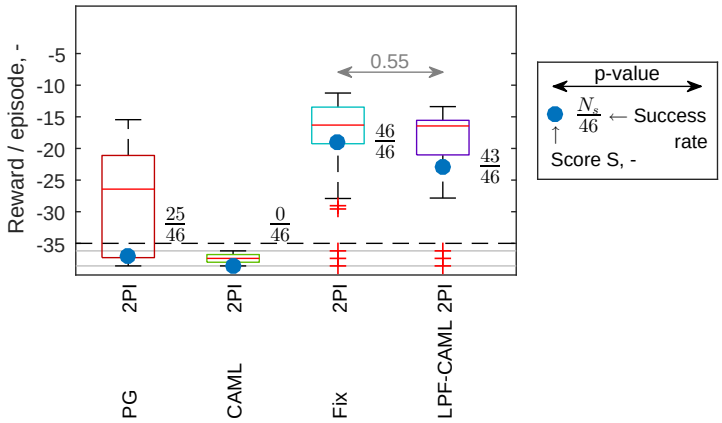


Figure 5.2: Performance of CAML trained with a Low-Pass Filter on the *Mixed LPV* test.

The modified CAML algorithm is also applied to the *Mixed LPV* test, as shown in Figure 5.2. The results are qualitatively similar to the results of the *Flapping* test. The original CAML and the PG algorithm are clearly outperformed by the modified CAML. The fixed controller achieves a similar tracking accuracy as the modified CAML, but a higher success rate. Compared to the *Flapping* test, the nonlinearity is an additional complication in the *Mixed LPV* test. It seems that the nonlinearity level of the DelFly does not lead to large performance decreases compared to the linear case. This is consistent with the findings of the *Nonlinearity* test.

5.2. Combination of algorithms

It was explained in Chapter 3 that a weakness of the Policy Gradient algorithms is that it requires a full episode with each setting. This is especially dangerous during the first runs, where large changes occur. CAML does not have this disadvantage, but may select unstable gains during the later episodes, due to imperfection in the classifier. A straightforward combination of these two algorithms uses CAML during the first episodes, and the PG algorithm (with the newly identified model) afterwards.

A thorough analysis of this combined algorithm would require optimization of the number of episodes with CAML. This section presents a first analysis, with the switching point placed in the middle: five episodes are performed with CAML, after which five episodes are performed with the PG algorithm. This method is tested on the *Tracking* test, which deals with variability only. Figure 5.3 shows the results of this analysis.

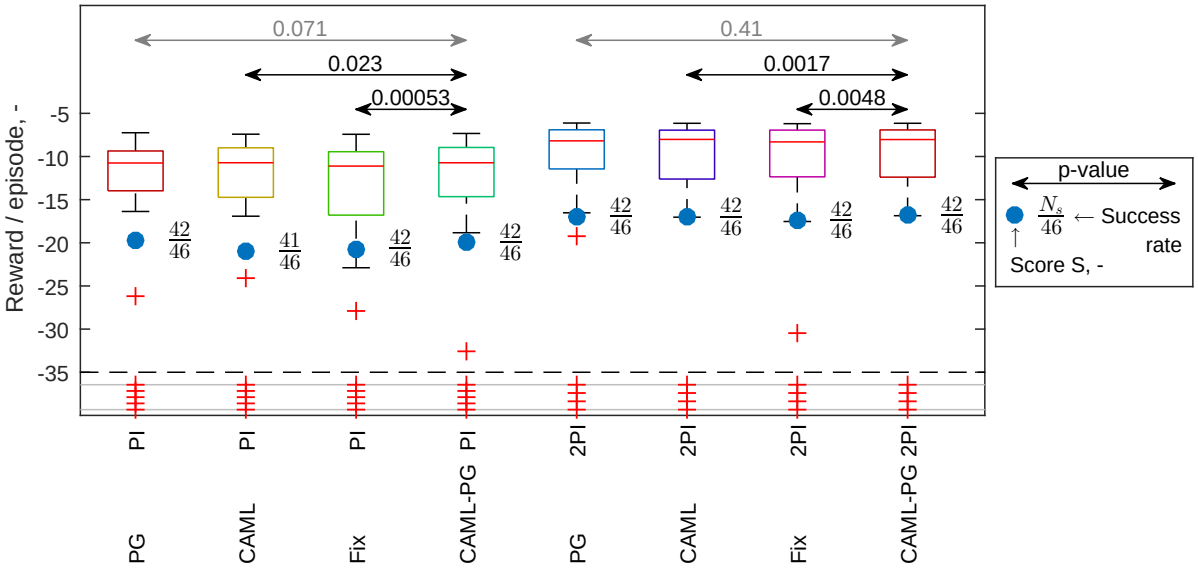


Figure 5.3: Performance of a combined algorithm on the *Tracking* test, using CAML during the first five episodes, and the PG algorithm afterwards.

For both single and double loop controllers, it is found that there is no significant difference between the original PG algorithm and the combined algorithm. There is a significant difference between the combined algorithm on one side, and CAML or the fixed controller on the other side. This is visible for the comparison between the single loop fixed controller and combined algorithm, but the boxplots appear to be very similar in the other cases. Further analysis of the test statistic of Wilcoxon's signed rank test [54] shows that this is due to individual differences on the different models, which cannot be represented in a boxplot. The combined algorithm is the better performing one in all these cases.

Optimization of the switching point may improve the results with the combined algorithm. In this analysis, it is found that the difference with the original PG algorithm is not significant. It is therefore expected that larger improvements can be made by analyzing the other recommendations.

6

Introduction to Reinforcement Learning

Flight control deals with continuous states and actions. However, Reinforcement Learning was originally developed for discrete systems. A good understanding of the discrete methods is required to fully comprehend the flight control methods in Chapter 7. This chapter introduces some discrete methods of Reinforcement Learning. Section 6.1 considers some basic concepts. Different methods are discussed in Section 6.2 to 6.3.

6.1. Basics

The goal of any controller is to produce an output based on an input. The most studied form of machine learning that performs this function is supervised learning [43]. In supervised learning, an agent is taught how to act by providing examples of good behavior. In terms of control engineering, this means that the intelligent controller will copy the behavior of an existing controller.

Reinforcement Learning is a general term for many algorithms that are capable of learning by trial and error [43]. These algorithms do not require an example controller: they learn the correct behavior by interacting with the system to be controlled. In this way, the controller can be made truly intelligent: it may invent methods of control that the designer had never considered.

6.1.1. Origins of Reinforcement Learning

This report considers Reinforcement Learning from a control engineering perspective. However, control engineers are not the only scientists interested in the topic. Sutton and Barto [43] highlight that Reinforcement Learning emerged from three different threads.

- The first thread that has lead to the modern methods is trial-and-error learning. This was initially investigated by scientists studying animal learning behavior, and it was also a topic of interest of the early research on Artificial Intelligence [43]. This is what distinguishes Reinforcement Learning from other types of learning and control.
- Reinforcement Learning is fundamentally related to optimal control. This field of research considers controllers that are optimal in some sense. In continuous Linear Time-Invariant control, the Linear Quadratic Regulator is an example of an optimal controller, which minimizes a quadratic cost function [37]. In general discrete-state control, one of the major concepts is dynamic programming, which is discussed in Section 6.2. It is important to realize that a Reinforcement Learning controller is always trying to optimize some function.
- The third discovery which is important to RL is Temporal Difference learning. This will be considered in Section 6.3.

6.1.2. Important concepts

In Reinforcement learning, the controller is called an *agent*, and the controlled system is called the *environment*. Their interaction is illustrated in Figure 6.1 The agent performs *actions*, which influence the *state* of the environment. The actions a_t of the agent depend on the current state s_t, according to a *policy* π. If the policy is deterministic, $a_t = \pi(s_t)$ present the action taken in state s_t. For a stochastic policy, $\pi(s_t, a_t)$ presents

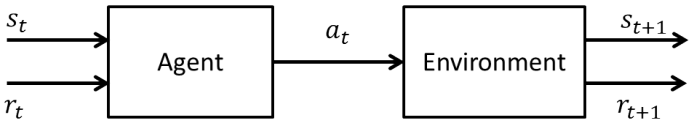


Figure 6.1: Interaction between agent and environment (inspired by [43]).

the chance of taking action a_t in state s_t. The change in the state of the environment may also be either deterministic or stochastic.

It was stated in the previous subsection that the algorithm is always trying to optimize something. For this, the *reward* function R is introduced. At every time step, the agent perceives a reward r_t for its previously performed actions. This reward originates from the environment, and may depend (deterministically or stochastically) on the current state, previous state and the previous action. The RL agent may discount later rewards exponentially by a factor γ. This leads to Eq. 6.1 for the *return* G.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{6.1}$$

The single goal of an RL agent is to maximize the return. The *value function* is defined as the expectation of the return function, when following policy π from the current state s, as shown in Eq. 6.2. This is a state-value function. Alternatively, an action-value function may be defined according to Eq. 6.3, as the expected return when performing action a and following π afterwards.

$$V^{\pi}(s) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right] \tag{6.2}$$

$$Q^{\pi}(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right] \tag{6.3}$$

When evaluating a value function, it is important that the state provides all relevant information. The process is said to have the *Markov* property if Eq. 6.4 is satisfied [43].

$$P(r_{t+1} = r, s_{t+1} = s' | s_t, a_t) = P(r_{t+1} = r, s_{t+1} = s' | s_0, s_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t) \tag{6.4}$$

Equation 6.4 means that the current state must provide all relevant information that would be available if the whole state history was known. Thus, the actor has all information required to make an optimal decision. The situation where an actor has to make the optimal decision with knowledge of a Markov state is called a *Markov Decision Process*. This is the problem for which most Reinforcement Learning theory was developed.

6.2. Dynamic Programming

Dynamic Programming (DP) is a method from optimal control to find the optimal policy. While it does not actually learn by trial and error, the concepts of this planning method are very important for RL.

6.2.1. Bellman equations

Dynamic Programming is based on the Bellman equation, which is found by expressing Eq. 6.2 in the recursive form shown in Eq. 6.5.

$$V^{\pi}(s) = E\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})\right] \tag{6.5}$$

For a given policy π, the value function must satisfy Eq. 6.5. When using DP, it is required to evaluate a value function for a given policy. This *policy evaluation* is done recursively by turning Eq. 6.5 into an update rule. This is shown in Eq. 6.6.

$$V^{\pi}_{k+1}(s) = E\left[r_{t+1} + \gamma V^{\pi}_{k}(s_{t+1})\right] \tag{6.6}$$

In the general case, the policy, state transition and reward may be stochastic. The probability distributions of all three are required to evaluate the expectation operator. This requires a complete model of the

Algorithm 6.1 Policy Iteration [43]

1: Initialize $V(s)$ and $\pi(s)$ randomly
2: **repeat**
3: **repeat** {Policy evaluation}
4: **for all** s **do**
5: $V(s) \Leftarrow \sum_{s'} p(s'|s,\pi(s)) \left[r(s,\pi(s),s') + \gamma V(s') \right]$
6: **end for**
7: **until** changes are small
8: **for all** s **do** {Policy improvement}
9: $\pi(s) \Leftarrow \arg\max_a p(s'|s,\pi(s)) \left[r(s,\pi(s),s') + \gamma V(s') \right]$
10: **end for**
11: **until** $\pi(s)$ is stable
12: **return** $\pi(s)$

environment. If the probability to go from s to s' with action a is denoted as $p(s'|s,a)$, Eq. 6.7 gives the update equation.

$$V^{\pi}_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left[r_{t+1} + \gamma V^{\pi}_k(s_{t+1}) \right] \tag{6.7}$$

For a given stochastic policy, this equation can be used repeatedly to determine the value function. In the next section, it will be explained how this can be used to solve the planning problem.

6.2.2. Generalized Policy Iteration

In itself, the value function is of no use. The importance of the value function is that it 'ranks' the states in terms of desirability, when following the policy π. This information is very useful when the goal is to determine a better policy. For a given value function, *policy improvement* can done by selecting the action that will result in the greatest return. This leads to the concept of Generalized Policy Iteration (GPI), where steps of policy evaluation and improvement are performed in some order.

The idea is that the value function and policy are successively updated. First, the value function is determined for a given policy. This is an iterative process, which may be halted at some stopping criteria. Next, a new policy is determined based on the value function. The new policy can be determined in several ways. Some examples are given below:

- **Random:** Random policy selection results in the greatest possible exploration in the state space, but does not make use of new information.
- **Greedy:** With greedy selection, the action is selected that results in the greatest value. This leads to maximum exploitation of knowledge. However, this may result in parts of the state space being unexplored, which may result in suboptimal behavior.
- ϵ**-Greedy:** This process usually selects the greedy action, but with probability ϵ, it selects a random action. A careful selection of ϵ makes it possible to balance exploration and exploitation.

These policy improvement options highlight the trade-off between exploration and exploitation. This trade-off is found very often in RL, and in optimization problems in general.

The most basic example of GPI is Policy Iteration, which is shown in algorithm 6.1 [43]. This algorithm first completes the policy evaluation in steps 3 to 7. For a given policy, the value function is determined by means of DP. Policy improvement occurs in steps 8 to 10. For every s, the greedy action is selected. This leads to a deterministic policy. For this new policy, the value function is again determined, and the policy is improved again. Policy Iteration can be proven to converge to an optimal policy [43].

Another GPI example is Value Iteration. This algorithm is almost the same as Policy Iteration, but only performs one step of policy evaluation for each new policy. This makes it possible to write the policy evaluation and improvement in one step, as shown in algorithm 6.2 [43]. The algorithm is still guaranteed to converge to an optimal policy [43].

Algorithm 6.2 Value Iteration [43]

1: Initialize $V(s)$ randomly
2: **repeat**
3: **for all** s **do**
4: $V(s) \Leftarrow \max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma V(s')\right]$
5: **end for**
6: **until** changes are small
7: $\pi(s) = \arg\max_a \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma V(s')\right]$
8: **return** $\pi(s)$

6.3. Temporal Difference methods

Just like Dynamic Programming, Temporal Difference (TD) methods make use of the Bellman equation (6.5). However, instead of computing the expectation operator by means of probability distributions, the estimate of the value function is changed by experience.

6.3.1. Temporal Difference Learning

The most simple form of TD learning updates directly in the direction of the Bellman equation. This is done according to Eq. 6.8.

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s)\right] \tag{6.8}$$

On every step, the value function is changed towards the experienced reward and the discounted value of the next state. Because the new estimate is based on other estimates (of $V(s_{t+1})$), it is called a *bootstrapping* method. TD learning can be applied on every time step: it is not required to wait for the entire episode to finish.

6.3.2. Sarsa

The Sarsa-algorithm extends TD learning to action-value functions. This makes it possible to perform control without any knowledge of the model. The concept of Sarsa is stated in Eq. 6.9.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \tag{6.9}$$

The name Sarsa refers to the events that are required for the update (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}) [43]. On every time step, the action for the next step is chosen by comparing the possible actions a_{t+1} in s_{t+1}. This can be done using a greedy or ϵ-greedy approach, or with different methods.

6.3.3. Q-learning

Another method of determining action-value functions is Q-learning. This method is demonstrated in Eq. 6.10.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \tag{6.10}$$

This algorithm is very similar to Sarsa, the only difference being that the maximum is taken for $Q(s_{t+1},a)$. The consequence of this is that Q-learning converges to the optimal action-value function for the environment, regardless of the policy that is being followed. This means that a highly exploring strategy may be chosen, while Q-learning still converges to the optimal strategy. Note that when a greedy action selection is performed, Sarsa and Q-learning are equal.

7

Reinforcement Learning in aerospace control

The previous chapter has elaborately introduced the field of Reinforcement Learning. This chapter will consider how this theory can be extended to deal with aerospace control problems. Section 7.1 will explain the possible solutions to these challenges. The remaining sections will elaborate on some promising solutions.

7.1. Continuous Reinforcement Learning solutions

The methods in Chapter 6 seem to be promising for the control of systems with discrete state-action spaces. However, they require a tabular storage of numerical data of the value function. For aerospace control problems, both the state and action spaces are typically continuous. This makes it impossible to use tabular storage.

Figure 7.1 shows an overview of the possible solutions to this problem. First of all, it possible to discretize the state and action spaces, such that any algorithm from Chapter 6 can be used. This is one of the easiest solutions, but it leads to problems when the continuous state-space is high-dimensional. In that case, the number of discrete states can become very large, leading to slow learning.

7.1.1. Extending Reinforcement Learning to continuous state-action spaces

A solution is to write the value function as a parametrized function of the state (and action). This is a very common solution in control engineering, and it commonly uses another notation. A comparison of the discrete and continuous notations is given in Table 7.1. The continuous state will be denoted by $x \in \mathbb{R}^n$, and the continuous action will be denoted by $u \in \mathbb{R}^m$. The symbol $J(x,u;\theta)$ is used for a continuous value function, with parametrization θ. This function can be dependent on the state only, or on both state and action.

In discrete RL systems, the reward function is defined for transition from t to $t+1$ as $r_{t+1} = r(s_t, a_t, s_{t+1})$. Hence, it depends on the transition from the previous state to the current state. In continuous RL, it is commonly found that the reward (often called utility) is independent of the state at $t+1$. In this report, the symbol U denotes the utility, where the notation $U_t = U(x_t, u_t)$ is used. Note that U_t and r_{t+1} refer to a similar transition, but use a different time index!

An important distinction to make is the difference between dynamic programming and temporal difference learning. Dynamic programming requires a complete model of the environment, which makes it more

Table 7.1: Discrete and continuous notations in Reinforcement Learning.

Parameter	Discrete	Continuous	Notes
State	s	x	s can be anything, $x \in \mathbb{R}^n$
Action	a	u	a can be anything, $u \in \mathbb{R}^m$. u is often called (control) input
Reward	r_{t+1}	U_t	Inconsistency in time index. U if often called utility
Value	$V(s)$	$J(x)$	
Action-value	$Q(s,a)$	$J(x,u)$	

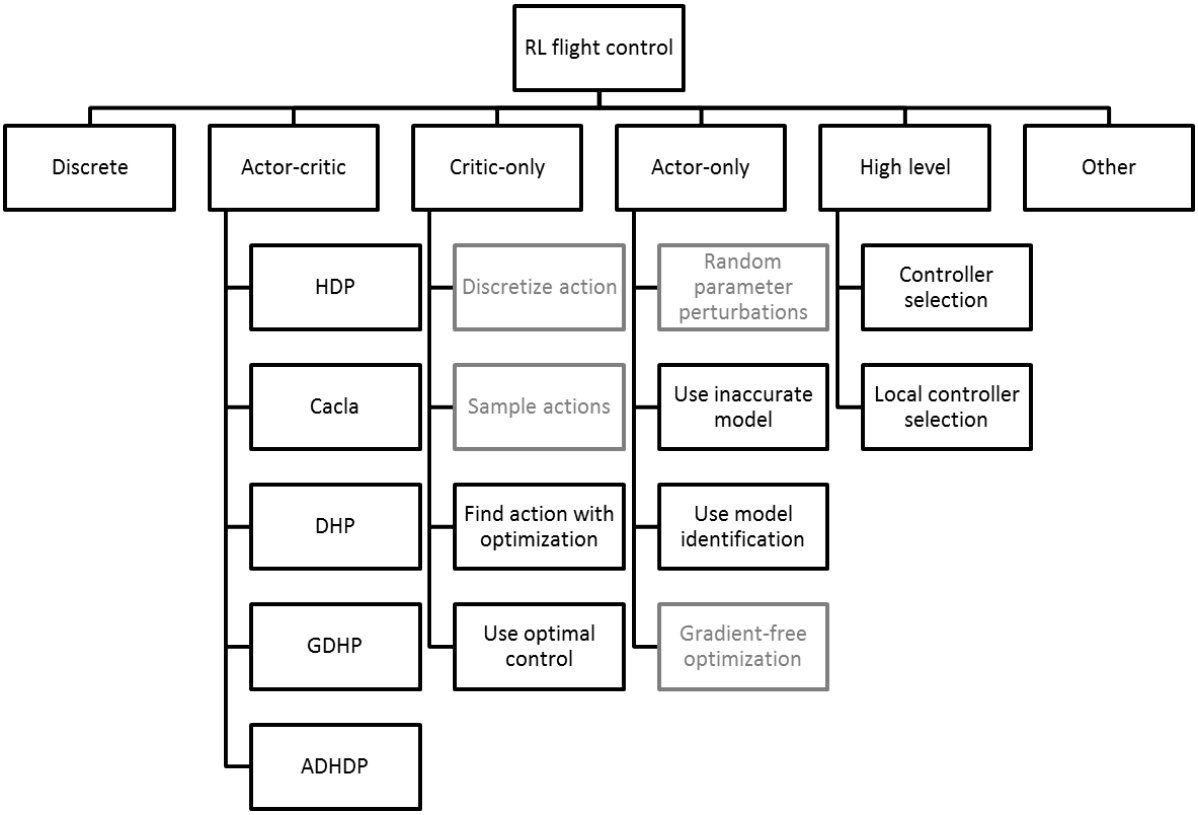


Figure 7.1: Overview of Reinforcement Learning solutions to flight control. Solutions in gray are not discussed this report, but included for completeness.

of a planning method than a learning method. When using temporal differences with Sarsa or Q-learning, a completely model-free learning approach is obtained. In continuous RL, an often used solution is to identify a model online, and use dynamic programming algorithms with these models. These methods transform dynamic programming into a true learning method.

7.1.2. The action selection problem

When starting from discrete RL control, the easiest approach is to parametrize the state-value function, and use dynamic programming algorithms. The difficulty in this approach is the (greedy) action selection, shown in Eq. 7.1.

$$u(x) = \max_u J(x,u;\theta) \tag{7.1}$$

In discrete systems, it is easy to select the best action, by looking at the resulting states for all possible actions. This is more cumbersome when the action set is continuous: the resulting optimization problem can be difficult and non-convex. Figure 7.1 shows some possible solutions to this problem. These methods are categorized as *critic-only* methods. They will be elaborated in Section 7.3. For explanatory and historical reasons, actor-critic approaches will be explained first in Section 7.2.

7.2. Actor-critic algorithms

Actor-critic approaches, also called Adaptive Critic Designs (ACDs), make use of a separate actor to determine the optimal action for a given state. This actor is also an approximated function, and is parametrized using ψ. ACDs follow the concept of Generalized Policy Iteration, explained in Chapter 6: it consists of steps of policy evaluation (critic updating) and policy improvement (actor updating). Some common ACDs will be discussed in the next subsections.

7.2.1. Heuristic Dynamic Programming

The most straightforward ACD is the Heuristic Dynamic Programming (HDP) algorithm. Its critic network provides an estimate of the expected return in a state. This is implemented with the function $J(x;\theta)$, which means that the critic network output is not dependent on the action selected. If the critic output is correct, the Bellman equation must hold, as shown in Eq. 7.2 [40].

$$J(x(t);\theta) = \sum_{k=0}^{\infty} \gamma^k U(t+k) = U(t) + \gamma J(x(t+1);\theta) \tag{7.2}$$

If the critic output is incorrect, there will be an imbalance in Eq. 7.2. The HDP algorithm is intended to minimize this imbalance, using the error function shown in Eqs. 7.3 and 7.4 [40]. The shorter notation $J(t) = J(x(t);\theta_i)$, where the dependence on the state and parameters is implicit, is used for convenience.

$$E_c(t) = \frac{1}{2}(e_c(t))^2 \tag{7.3}$$

$$e_c(t) = J(t) - \left[\gamma J(t+1) + U(t)\right] \tag{7.4}$$

The common way to do this is by means of *backpropagation*. This method updates the parameters with a gradient descent algorithm to minimize the error, as shown in Eq. 7.5 [40]. Using the chain rule, the derivative[1] can be expressed as in Eq. 7.6.

$$\theta_{i+1} = \theta_i - l_c(t)\left[\frac{\partial E_c(t)}{\partial \theta_i}\right]^T \tag{7.5}$$

$$\frac{\partial E_c(t)}{\partial \theta_i} = \frac{\partial E_c(t)}{\partial e_c(t)}\frac{\partial e_c(t)}{\partial J(t)}\frac{\partial J(t)}{\partial \theta_i} = e_c(t)\frac{\partial J(t)}{\partial \theta_i} \tag{7.6}$$

Note that the method neglects the dependence of $J(t+1)$ on θ_i, and treats the term between brackets in Eq. 7.4 as constant [40]! This is in agreement with the discrete dynamic programming methods in Chapter 6. This idea will be discussed in more detail in Chapter 8. The resulting method is able to optimize the parameters θ to find the value function of a given policy.

The actor has as its goal to maximize the sum of future utility. It uses the critic network to estimate this sum. The optimization is again performed using backpropagation, as shown in Eq. 7.7. Since $J(t)$ is not directly dependent on the actions and action parameters, the Bellman equation is used to find a backpropagation path from $J(t+1)$ through the model network, as shown in Eq. 7.8. This requires knowledge of the dependence of the utility function on the actions. In literature [40, 47], it is common to use action-independent reward functions, such that $\frac{\partial U(t)}{\partial \psi_i} = 0$, and $J(t+1)$ can be optimized directly.

$$\psi_{i+1} = \psi_i + l_a(t)\left[\frac{\partial J(t)}{\partial \psi_i}\right]^T \tag{7.7}$$

$$\frac{\partial J(t)}{\partial \psi_i} = \frac{\partial U(t)}{\partial \psi_i} + \gamma\frac{\partial J(t+1)}{\partial \psi_i} = \frac{\partial U(t)}{\partial u(t)}\frac{\partial u(t)}{\partial \psi_i} + \gamma\frac{\partial J(t+1)}{\partial x(t+1)}\frac{\partial x(t+1)}{\partial u(t)}\frac{\partial u(t)}{\partial \psi_i} \tag{7.8}$$

Performing the actor update requires the derivatives of the value function to the state, which are found by backpropagation through the critic network. Also, a model is required to determine the derivative $\frac{\partial x(t+1)}{\partial u(t)}$. The implementation in [46] performs online system identification to find this model.

A variation to this is formed by the Continuous Actor-Critic Learning Automaton (Cacla) [45]. This method uses the same manner of updating the critic. The action u is selected as the action suggested by the actor (u_a) plus a random variation. The actor update then uses the TD error in the critic. Using Eq. 7.4, one can conclude

[1] In this report, consistent numerator layout will be used for vector and matrix derivatives. For a $n \times 1$ vector y and a $m \times 1$ vector x, this means that:

$$\frac{dy}{dx} = \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \cdots & \frac{dy_1}{dx_m} \\ \frac{dy_2}{dx_1} & \frac{dy_2}{dx_2} & \cdots & \frac{dy_2}{dx_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dy_n}{dx_1} & \frac{dy_n}{dx_2} & \cdots & \frac{dy_n}{dx_m} \end{bmatrix}$$

that an action was unexpectedly successful if $e_c(t) < 0$. This knowledge is used by Cacla to result in the update rule in Eq. 7.9 [45].

$$\psi_{i+1} = \begin{cases} \psi_i + l_a(t)(u(t) - u_a(t)) \frac{\partial u_a(t)}{\partial \psi_i} & e_c(t) < 0 \\ \psi_i & e_c(t) > 0 \end{cases} \tag{7.9}$$

Cacla only updates the actor if it performs unexpectedly well. Also, the update size is not proportional to the TD error. This has advantages if exploratory actions are taken [45]. Cacla is considered a more exploratory variation to HDP in this report. An advantage is that it requires no model. It is expected that its performance is close to HDP. This algorithm will be revisited if HDP is successful.

7.2.2. Dual Heuristic Programming

To the controller, the critic network is merely a means to update the actor. A careful look at Eq. 7.8 shows that the HDP algorithm does not use only the critic to update the actor: it also uses the partial derivatives of the critic to the states.

The Dual Heuristic Programming (DHP) algorithm estimates these derivatives directly. With more accurate estimates of these derivatives, it should be possible to perform better updates of the actor.

DHP defines the *costate* vector $\lambda(t) = \left[\frac{\partial J(t)}{\partial x(t)}\right]^T$, which is an $n \times 1$ vector in the numerator layout of this report. The DHP critic estimates the costate, rather than the cost function $J(t)$.

It was found that the DHP critic can lead to better results than the HDP critic, because it approximates the costate directly [40]. However, it is more complex, because it has to estimate a vector instead of a scalar. Globalized Dual Heuristic Programming (GDHP) is an attempt to combine the best of both approaches [40]. The GDHP critic approximates the value function, but its update process is such that its derivatives are also optimized. Prokhorov and Wunsch II [40] showed that the performance of GDHP is very similar to that of DHP. However, it has not been applied as rigorously to aircraft.

7.2.3. Action Dependent Heuristic Dynamic Programming

A large disadvantage of the previous two methods is that they require a model of the environment. This is because they are based on Dynamic Programming. The model is required for the backpropagation path between the critic and the actor. By using the action as a second input of the critic network, it should be possible to perform backpropagation without the model network. If this is done with the HDP critic, Action Dependent Heuristic Dynamic Programming (ADHDP) results.

When the action is also an input of the critic, the method becomes very similar to Q-learning. Even if the utility function is not directly dependent on the action, the action influences the next state and therefore has an effect on the expected return. However, this means that the model will be 'absorbed' by the critic network [28, 46].

Si and Wang [42] introduced a method that uses ADHDP with a backwards TD scheme, which allows model-free control. This method has been applied rigorously to the Apache helicopter [18], under the name of Direct Neural Dynamic Programming (DNDP).

Liu [28] highlights that with a slightly different definition of the value function, this method becomes very similar to HDP. The value function $\mathfrak{Q}$ is defined according to Eq. 7.10.

$$\mathfrak{Q}(t) = \sum_{k=1}^{\infty} \gamma^{k-1} U(t+k) \tag{7.10}$$

This new value function it the same as the previously used $J(t)$, but disregards the reward at the current time step: $\mathfrak{Q}(t) = J(t+1)$. Several applications using ADHDP make use of this value function [18, 42]. The Bellman equation for the $\mathfrak{Q}$-function is then given by Eq. 7.11.

$$\mathfrak{Q}(t) = \gamma \mathfrak{Q}(t+1) + U(t+1) \tag{7.11}$$

In ADHDP, the actor updating occurs according to Eqs. 7.14 to 7.15. In Eq. 7.15, the J-function has been substituted for the $\mathfrak{Q}$-function.

$$E_a(t) = \frac{1}{2}(e_a(t))^2 \tag{7.12}$$

$$e_a(t) = \mathfrak{Q}(t) - b(t) \tag{7.13}$$

$$\psi_{i+1} = \psi_i - l_a(t)\left[\frac{\partial E_a(t)}{\partial \psi_i}\right]^T \tag{7.14}$$

$$\frac{\partial E_a(t)}{\partial \psi_i} = \frac{\partial E_a(t)}{\partial e_a(t)}\frac{\partial e_a(t)}{\partial \mathfrak{Q}(t)}\frac{\partial \mathfrak{Q}(t)}{\partial u(t)}\frac{\partial u(t)}{\partial \psi_i} = e_a(t)\frac{\partial J(t+1)}{\partial u(t)}\frac{\partial u(t)}{\partial \psi_i} \tag{7.15}$$

HDP requires two functions to estimate $J(t+1)$, while ADHDP only requires one. The two functions in HDP could be seen as a two-layer function which performs the same computation [28]. However, a difference is that the model layer is trained to match the dynamics, while the critic layer is trained to find the value function. Another difference is that the ADHDP implementation considered in this section uses a backward updating process. This will be discussed in Chapter 8.

The action dependence can also be introduced in the DHP network. In that case, however, a model is still required for backpropagation [40]. This is therefore not considered desirable.

7.2.4. Comparison of actor-critic algorithms

An extensive comparison between different ACDs has been performed by Prokhorov and Wunsch II [40]. On an automatic landing system for a simplified commercial aircraft, several ACDs were implemented. The goal of the controller is to land the aircraft under a wind shear condition subject to gusts. Hence, this is both a tracking and a disturbance rejection task. It is found that GDHP and DHP lead to the highest success rates. Lower success rates are obtained with HDP. Venayagamoorthy et al. [47] found that DHP outperforms HDP on a neurocontroller for a turbogenerator connected to the power grid. Werbos [52, 53] also states that DHP is a more advanced and better algorithm.

A disadvantage of DHP is its learning speed. Because a vector needs to be estimated, the learning process can take much longer. For the automatic landing system, it was found that DHP takes ten times as many trials to learn as HDP [40]. Also, its larger complexity increases the computational burden [27]. However, Werbos [52] states that DHP can be much faster, on the condition that the correct model is known.

In this case, the controller should function as an adaptive controller. It is expected that the learning speed, when started from a random initialization, provides a good indication of the adaptation speed of the controller [46]. This adaptation should be robust to model uncertainties. This is the reason why HDP is preferred over DHP in this project.

The possibility of model-free control makes ADHDP an attractive option. Since it has been applied in detail to the Apache helicopter [18], it is expected that a good controller can be obtained for the DelFly. For the autolanding problem, however, it was found that ADHDP leads to much lower success rates than HDP [40]. This may be due to the exact implementation of the algorithm [42]. Van Kampen et al. [46] compared HDP and ADHDP directly, and found that HDP outperforms ADHDP as an adaptive controller for the F-16 aircraft. While ADHDP was found to be more robust to noise, the HDP controller seemed to be more applicable to other flight conditions. Because of the similarity with the DelFly case, HDP is considered a more suitable controller for the DelFly.

7.3. Critic-only algorithms

It was explained in Section 7.1 that the problem with extending discrete RL to continuous state-action spaces is the action selection. The actor-critic algorithms were the first solution to this problem. More recently, some other solutions have become available. This section will analyse algorithms that use only a critic network. Figure 7.1 contains an overview of the possible solutions.

7.3.1. Solving the optimization problem

It was stated in Section 7.1 that the action selection can be a complex optimization problem. If a model is available, a possibility is to solve this problem using any optimization method, such as Newton's method or Nelder-Mead optimization [36]. If the action-value function is approximated by a sufficiently differentiable function approximator, gradients and Hessians are available analytically. While this optimization leads to an increased computational burden and the proposed methods use local optimization algorithms, the resulting

algorithms are more straightforward than ACDs. A case study considering the swing-up of the Acrobot[2] [36], however, it was shown that the actor-critic-based Cacla outperforms the critic-only methods presented in this subsection.

7.3.2. Including the optimal control

In ACDs, the actor is aimed at optimizing the return. The actor is optimized by considering the critic function (and sometimes the model). It was explained in Chapter 6 that Reinforcement Learning originated from the field of optimal control. Optimal control theory provides methods to calculate optimal actions analytically. Such theory can be used to replace the actor network in ACDs.

An important intermediate step was performed by Ferrari and Stengel [19], who applied DHP methodology to the control system of a business jet. However, they changed the method of action selection. While the 'conventional' DHP first selects an action, and then updates the actor network, this method performs optimization before action selection.

The key idea is that the action must be optimal with respect to the value function: $\frac{\partial J(t)}{\partial u(t)} = 0$. This can be elaborated by means of the Bellman equation as shown in Eq. 7.16 [3].

$$\frac{\partial J(t)}{\partial u(t)} = \gamma \frac{\partial J(t+1)}{\partial u(t)} + \frac{\partial U(t)}{\partial u(t)} = \gamma \lambda^T(t+1) \frac{\partial x(t+1)}{\partial u(t)} + \frac{\partial U(t)}{\partial u(t)} = 0 \tag{7.16}$$

Ferrari and Stengel [19] computed the optimal control by solving Eq. 7.16 iteratively. The output of the actor network is used as a first guess, and Newton's method is used to improve this estimate iteratively. When the optimal control is found, it is applied to the system, and the actor network is trained towards the optimal control. Note that a model of the system is required to train the actor.

7.3.3. Single Network Adaptive Critic

Padhi et al. [38] proposed an optimal control-based RL method that completely removes the need for the actor network. Their Single Network Adaptive Critic (SNAC) method requires two functions: the critic and the model. Only the critic is updated online in this implementation. The method is applicable to systems where the optimal control can be computed analytically from the state and the costate. This includes input-affine discrete-time systems of the form of Eq. 7.17 with a quadratic utility function, shown in Eq. 7.18.

$$x(t+1) = f(x(t)) + g(x(t))u(t) \tag{7.17}$$

$$U(t) = \frac{1}{2}\left[x^T(t)Qx(t) + u^T(t)Ru(t)\right] \tag{7.18}$$

The SNAC method is very similar to DHP. It is based on two equations. The optimal control equation is presented in Eq. 7.19. Using the structure of Eqs. 7.17 and 7.18, this reduces to Eq. 7.20, which can be solved for u as in Eq. 7.21 [39]. Note that R (as well as Q) is a symmetric matrix. This equation[3] provides an explicit expression for the optimal control at time t.

$$\gamma \lambda^T(t+1) \frac{\partial x(t+1)}{\partial u(t)} + \frac{\partial U(t)}{\partial u(t)} = 0 \tag{7.19}$$

$$\gamma \lambda^T(t+1) g(x(t)) + u(t)^T R = 0 \tag{7.20}$$

$$u(t) = -\gamma R^{-1}\left[g(x(t))\right]^T \lambda(t+1) \tag{7.21}$$

The costate equation is obtained by setting the DHP critic error to zero [39], as shown in Eq. 7.22. This is rephrased in Eq. 7.23. Under the condition that the optimal control is taken, Eq. 7.16 can replace the term between brackets in Eq. 7.23 to arrive at Eq. 7.24. As a final step, the **known** model in Eq. 7.17 can be used to substitute $\frac{\partial x(t+1)}{\partial x(t)}$, resulting in Eq. 7.25.

[2] The Acrobot is a double pendulum, consisting of two links. Only the hinge between the two links is actuated. The controller task is to swing up the Acrobot from its neutral position [36].

[3] This method was originally presented for undiscounted settings ($\gamma = 1$). It has been extended to the discounted setting for consistency with the previous parts.

Algorithm 7.1 Training algorithm for the SNAC approach [39]

for $i = 1, 2, ..., I$ **do**
  repeat
    Generate a subset S_i of initial states $x(0)$
    for all $x(0) \in S_i$ **do**
      Input $x(0)$ to the critic to obtain $\lambda(1)$
      Calculate the optimal control $u(0)$ from Eq. 7.21
      Get $x(1)$ from the model Eq. 7.17
      Input $x(1)$ to the critic to obtain $\lambda(2)$
      Insert $\lambda(2)$ into the costate Eq. 7.25 to calculate a desired $\lambda_d(1)$
    end for
    Train the network towards $\lambda_d(1)$ for all $x(0) \in S_i$
  until convergence of the critic network
end for

$$\frac{\partial J(t)}{\partial x(t)} = \frac{\partial}{\partial x(t)}\left[\gamma J(t+1) + U(t)\right] = \gamma \frac{\partial J(t+1)}{\partial x(t+1)}\left[\frac{\partial x(t+1)}{\partial x(t)} + \frac{\partial x(t+1)}{\partial u(t)}\frac{\partial u(t)}{\partial x(t)}\right] + \frac{\partial U(t)}{\partial x(t)} + \frac{\partial U(t)}{\partial u(t)}\frac{\partial u(t)}{\partial x(t)} \tag{7.22}$$

$$\frac{\partial J(t)}{\partial x(t)} = \gamma \frac{\partial J(t+1)}{\partial x(t+1)}\frac{\partial x(t+1)}{\partial x(t)} + \frac{\partial U(t)}{\partial x(t)} + \left[\gamma \frac{\partial J(t+1)}{\partial x(t+1)}\frac{\partial x(t+1)}{\partial u(t)} + \frac{\partial U(t)}{\partial u(t)}\right]\frac{\partial u(t)}{\partial x(t)} \tag{7.23}$$

$$\lambda^T(t) = \gamma \lambda^T(t+1)\frac{\partial x(t+1)}{\partial x(t)} + x^T(t)Q \tag{7.24}$$

$$\lambda^T(t) = \gamma \lambda^T(t+1)\left[\frac{\partial f(x(t))}{\partial x(t)} + \frac{\partial g(x(t))}{\partial x(t)}u(t)\right] + x^T(t)Q \tag{7.25}$$

The key to the SNAC algorithm is that the critic uses $x(t)$ to approximate $\lambda(t+1)$. This makes it possible to obtain optimal control inputs without requiring an actor. Padhi et al. [39] propose algorithm 7.1 for training of the critic, making use of Eq. 7.26.

$$\lambda_d(1) = \gamma \lambda(2)\left[\frac{\partial f(x(t))}{\partial x(t)} + \frac{\partial g(x(t))}{\partial x(t)}u(t)\right]^T + Qx(t) \tag{7.26}$$

$$\lambda_d(1) = \gamma \lambda(2)\left[\frac{\partial f(x(t))}{\partial x(t)} + \frac{\partial g(x(t))}{\partial x(t)}u(t)\right]^T + Qx(t) \tag{7.27}$$

This algorithm uses batch updating of a set of states S_i, until the critic converges correctly for this batch. Then, a new batch of states is used. Padhi et al. [39] suggest to generate a batch of random states within a hypersphere of a certain radius, and to increase this radius for each new batch. This updating process is limited to offline training. However, stochastic (one-by-one) updating can also be used with this algorithm, and is more suitable for online use. These updating methods will be compared in Chapter 8.

The SNAC algorithm is closely related to DHP, and suffers from the same disadvantages. Ding et al. [15] proposed J-SNAC, which uses the HDP critic to perform the same function. The algorithm is very similar to SNAC, and is shown in algorithm 7.2. It uses the same equations as the SNAC algorithm, and additionally uses the Bellman equation (restated in Eq. 7.28).

$$J_d(0) = \gamma J(1) + U(t) \tag{7.28}$$

This critic has the advantage that its output is a scalar. This is said to lower the amount of training required for the critic [16] and the memory requirements. Also, the J-SNAC output has a physical meaning [15], which makes it easier to interpret. The outputs of the SNAC network could have different orders of magnitude, which results in convergence problems [14].

The approach as described above requires a model of the environment to function. The same can be said of DHP and HDP, but HDP has been shown [46] to function with an online model updating. A version with online model updating, based on Lyapunov theory, has been developed for both SNAC [6] and J-SNAC [14].

Algorithm 7.2 Training algorithm for the J-SNAC approach [15]

for $i = 1, 2, ..., I$ **do**
  repeat
    Generate a subset S_i of initial states $x(0)$
    for all $x(0) \in S_i$ **do**
      Input $x(0)$ to the critic to obtain $J(0)$
      Use backpropagation to calculate $\lambda(0) = \frac{\partial J(0)}{\partial x(0)}$
      Insert $\lambda(0)$ into the costate Eq. 7.25 to obtain $\lambda(1)$
      Calculate the optimal control $u(0)$ from Eq. 7.21
      Get $x(1)$ from the model Eq. 7.17
      Input $x(1)$ to the critic to obtain $J(1)$
      Insert $J(1)$ into the Bellman Eq. 7.28 to calculate a desired $J_d(0)$
    end for
    Train the network towards $J_d(0)$ for all $x(0) \in S_i$
  until convergence of the critic network
end for

Comparisons between DHP and SNAC on a Van der Pol's oscillator and a MEMS actuator shows that the algorithms result in similar performance, but that SNAC requires slightly over half the computational load [39]. A direct comparison between HDP and J-SNAC gave a similar result [15]. From this, it is concluded that SNAC can be advantageous compared to ACDs. Its simpler implementation is another advantage. In a review by Wang et al. [49], SNAC is indeed considered an improvement over ACDs. However, (J-)SNAC is in a less mature stage. This research includes a direct comparison of HDP and J-SNAC, which is presented in Chapter 9.

7.3.4. Model-free critic-only control

In very recent work, Luo et al. [29] describe a form of Q-learning that is applicable to continuous state-action spaces. Their Critic-Only Q-Learning (CoQL) algorithm is model-free and can be used for tracking, even if the required path is not available in advance. Just-like Q-learning, it is an off-policy method, which can learn the optimal policy while following another policy. It can be used for any discrete-time nonlinear system; the system does not have to be input-affine. It uses least-squares techniques to estimate the critic. The optimal control input is calculated incrementally according to Eq. 7.29.

$$u(t) = u(t-1) + L \frac{\partial J(x(t), \tilde{u}}{\partial \tilde{u}}\bigg|_{\tilde{u}=u(t-1)} \tag{7.29}$$

Hence, the control is updated from the previous control without considering the change in state. Although this may seem strange, the algorithm was proven to converge to the optimal value function [29]. It is tested on a highly nonlinear two-state model. The critic weights seem to stabilize after approximately 40 iterations. This makes this a very promising method for the control of the DelFly. Its disadvantage is its novelty: it has never been tested on any system related to aerospace. Also, the critic must be pre-trained offline. Online updating is not practically feasible due to the iterative critic calculations. This is the reason why CoQL was not analyzed in detail during this project. However, informal analysis shows that the critic are relatively robust to changes in the system. Future research should take into account this algorithm.

7.4. Actor-only algorithms

A very different approach is to remove the value function entirely. In actor-only approaches, there is only a policy, parametrized using ψ. The tasks are necessarily episodic: TD errors cannot be used because there is no value function. These parameters are optimized to maximize some objective function F.

The task of improving the parameters becomes a conventional optimization problem, which can be solved using any optimization method. The critical aspect of actor-only methods is the execution of this optimization. For online learning, it should be considered that some policies can crash the DelFly. Therefore, small steps from known controller settings should be taken. Gradient-based optimization is one solution, shown in Eq. 7.30.

Algorithm 7.3 Actor-only approach using inaccurate models [1]

Input: T, with a locally optimal parameter setting ψ_+
Output: ψ

repeat
  $\psi \leftarrow \psi_+$
  Execute the policy with ψ in the real system, and record the state-action trajectory $x(0)$, $u(0)$, $x(1)$, $u(1)$, …, $x(n)$, $u(n)$
  for $t = 0, 1, 2, \ldots, n-1$ **do** {Construct a time-varying model $f_t(x,u)$ from $\hat{T}(x,u)$ and the trajectory}
    $f_t(x,u) \leftarrow \hat{T}(x,u) + \left[x(t+1) - \hat{T}(x(t),u(t))\right]$
  end for
  Use a local policy improvement algorithm on $f_t(x,u)$ to find a policy improvement direction d
  Determine the new parameters using $\psi_+ \leftarrow \psi + \alpha d$. Perform a line search in the real system to find α.
until ψ is not improved by the line search

$$\psi_{i+1} = \psi_i + l(t)\nabla_\psi F \tag{7.30}$$

Determining the gradient is one of the key aspects of these methods. While ACDs can use the critic to determine a gradient estimate, this is not possible for actor-only networks. This makes it rather complicated to apply such algorithms for online learning. Actor-only algorithms intended for online learning typically make use of a limited number of parameters, in order to reduce the difficulties of determining the gradient.

7.4.1. Making use of inaccurate models

Determining the gradients is much easier if a model is available during online control. Trial runs can be performed in simulation, while only using the best settings are used in the real system. This reduces the number of policies that are evaluated online, and thereby reduces the probability of using a dangerous policy. However, the model has to be very accurate for this to work correctly.

Abbeel et al. [1] proposed an algorithm which uses an inaccurate model to suggest parameter changes, and uses the real system to obtain the cost F. Algorithm 7.3 [1] shows a detailed implementation of this method. The real system dynamics are represented by Eq. 7.31. This is approximated by the inaccurate model in Eq. 7.32.

$$x(t+1) = T(x(t),u(t)) \tag{7.31}$$

$$\hat{x}(t+1) = \hat{T}(x(t),u(t)) \tag{7.32}$$

The algorithm uses the policy improvement direction d, rather than the gradient. This is a more general method, since it allows the algorithm to look further than the direction of the steepest descent, which could enable the algorithm to escape from local minima. However, using the gradient in the place of d is the most straightforward implementation.

An important step in this algorithm is the construction of the time-varying model $f_t(x,u)$. $\hat{T}(x,u)$ provides a time invariant representation of the system. However, it will (in general) not match the experienced trajectory exactly. By including the time-varying bias term $\left[x(t+1) - \hat{T}(x(t),u(t))\right]$ at every time step, $f_t(x,u)$ will match the experienced trajectory. The gradient obtained from $f_t(x,u)$ is then the gradient on the experienced trajectory. This step can also be interpreted as sampling the disturbances during the test run.

For this algorithm, convergence to a local optimum can be proven if the local improvement algorithm is policy gradient [1]. However, this is only applicable if the true dynamics $T(x,u)$ are deterministic. Abbeel et al. [1] state that the algorithm is also applicable to systems that are close to deterministic. Considering the noisy state estimation of the DelFly, a high sensitivity to noise is problematic.

Junell et al. [21] applied algorithm 7.3 to the control of a linearized a model of the F-16 aircraft. For the inaccurate model, a uniformly distributed error of 45% was introduced to every matrix element in the state equation. Even with such a large error, it was found that the algorithm finds a near-optimal policy after only sixteen trials[4] [21].

[4] Apart from the algorithm, the simple function approximator in this work may also contribute to the high learning speed.

The algorithm was also applied to a quadrotor [21]. No more than five trials were required to find a near-optimal policy in simulation [21]. Interestingly, quadrotor flight tests were performed in the Delft University of Technology Cyber Zoo, which will also be used for the DelFly. When starting from the default gains, it was found that the quadrotor finds its optimal performance after three trial and hits its stopping criterion (which is slightly different than in algorithm 7.3) after five trials [21]. However, the resulting performance is lower than in simulation.

The rapid learning and flight-proven status makes this algorithm very promising for the DelFly. The possibility of applying the algorithm to the DelFly is even mentioned explicitly by Junell et al. [21]. However, the more complicated dynamics, which are known less well, make the DelFly application more difficult. Since the same optical tracking equipment is used, the noise levels for the DelFly test will be of the same order of magnitude. However, it is expected that the behavior of the DelFly will be more random, for several reasons:

- The IMU in the DelFly is expected to be of lower accuracy.
- The lower weight of the DelFly makes it more susceptible to external disturbances.
- The DelFly is much smaller than the quadrotor considered. For the same accuracy in position determination, this leads to a lower accuracy in attitude determination. However, the size of the DelFly is considered large enough to avoid real problems due to this issue.

Another demonstration of using policy gradient methods is provided by Lupashin et al. [30]. This method applies control on a higher level and uses a different algorithm, but also uses a gradient-based method with a limited number of parameters. In a different optical tracking chamber, a quadrotor is controlled to perform double and triple flips. For double flips, the performance seems to stabilize after 40 to 50 iterations [30]. This provides additional evidence that policy gradient methods allow rapid learning, even in stochastic environments. This seems to be a very promising algorithm, which should be investigated in more detail.

7.4.2. Stochastic model identification

In the previous subsection, it was discussed that an inaccurate model can be used in the actor improvement step. A logical continuation is to improve the model during operation.

This subsection will first discuss one related method, in order to ensure a logical flow to the coming methods. Ng et al. [35] applied RL control to inverted helicopter flight, where the controller learns to balance a helicopter upside-down. A stochastic model is first identified while the helicopter is controlled by a human pilot. Next, the model is frozen: it is not updated online. A gradient-based algorithm is then used to find the optimal policy in the simulation model. This policy is finally applied to the real problem. This method uses a fixed policy, and should therefore be classified as a robust method, rather than an adaptive method.

At the level at which the approach is discussed in this report, it seems quite straightforward. Nevertheless, this approach is successful in balancing a helicopter upside-down, during outdoor flight tests. Two aspects of this approach are considered important:

- The real helicopter is controlled using a fixed policy. Although outdoor flight tests are subject to disturbances and the model is imperfect, this single policy is robust enough to perform its task.
- The training model is stochastic. It makes sense that a stochastic training process is required to deal with stochastic systems. The stochastic model of the helicopter assumes Gaussian errors for the one-step prediction [35].

An approach with online updating was introduced by Deisenroth and Rasmussen [13] under the name Probabilistic Inference for Learning Control (PILCO). This method calculates a stochastic model of the system during a trial with a given policy. This model is then used to update the policy, followed by a new step of policy evaluation and model updating. Interestingly, PILCO uses an analytical method to calculate the gradients of the stochastic model. PILCO has been shown to have a high data efficiency for real-life cart-pole systems [13] and helicopter hovering [31]. Ha and Yamane [20] developed a similar, but simpler method. Unfortunately this latter method has never been applied to aerospace systems. Tedious applications to non-aerospace systems are also not available.

Online updating of stochastic models seems promising in literature, but stochastic models will be difficult to identify, because they must capture the spread in the data. Also, this requires a valid stochastic model structure. System identification should not be the focus of this project. It is therefore decided to focus on the other options first. Stochastic model updating should be revisited if other options do not give acceptable results.

7.5. High-level control methods

In the previous sections, the 'intelligence' of the RL control method was located at the lowest possible level. A different approach is to use intelligent control on a higher level, where RL is used to select a pre-defined controller. These methods make use of predefined controllers, and are therefore classified as robust methods.

7.5.1. Helicopter hovering competitions

A lot of research has been performed on applying high-level methods to helicopter hovering control. This was largely driven by the Reinforcement Learning Competitions[5] in 2008, 2009, 2013 and 2014. Helicopter hovering was one of the domains of these competitions.

The competitions were performed entirely in simulation. The task in the competitions is to control an unknown three-dimensional model of a helicopter to stay close to a desired position. The model has twelve states, and the action vector is four-dimensional. The complication is that the model is altered due to wind. The wind is drawn from a random distribution that is unknown to the participants [24].

For the 2008 competition, participants were supplied with ten simulation models of the helicopter, drawn from the random distribution. However, the participants were not told how the variations were distributed. The RL controller had to be trained on these ten models. The competition was performed on fifteen new models, originating from the same distribution. Thus, the RL controller had to be able to generalize the ten available models to the entire distribution. During the test of the competition, the controller was evaluated on 1000 episodes of ten minutes each, for fifteen models. The 2009 competition was comparable, but used different methods of randomizing the model. The exact setup of the test also varies, but it is always aimed at generalizing the RL controller from a limited number of training examples.

Koppejan and Whiteson [24] performed an extensive analysis of the 2008 and 2009 competitions. For the 2008 competition, they evaluated several model-based approaches. They won the competition with a very simple model-free approach. For each available model, an evolutionary method was used to find the optimal parameters of the controller. This results in ten controller options. The test used different models, for which the ten options were not optimal. For each unknown simulation model in the test, 1000 episodes had to be performed. The winning approach uses the first ten episodes to evaluate the performance of the ten controller options. The best performing controller was used for the remaining 990 episodes [24].

This approach is exceptionally simple. The 'intelligence' is applied only during the first ten episodes for each model. The fact that it won the competition (without ever crashing) provides evidence that robust high-level approaches may be more suitable than adaptive low-level controllers. However, this approach seems limited to low-noise systems, since noise would complicate controller selection. In this approach, the long episodes are used to overcome this issue.

After the 2008 competition, Koppejan and Whiteson [24] continued their analysis in order to further improve and generalize performance. A hybrid approach was proposed for the more general 2009 competition. From the controllers for the ten training models, the two most robust[6] controllers were selected. Two out of 1000 test episodes were spent on evaluating these controllers. During these episodes, a model was identified. This model is used to identify a new optimal policy. Three optimization trials were performed and the three resulting controllers were evaluated for one episode each. The best performing controller from the five evaluated options was used for the remaining 995 runs. Additionally, a safeguard was implemented to switch to a safe baseline policy when a crash is imminent. This results in a crash probability of approximately 1% [24].

The helicopter hovering competitions are interesting because of their similarities to the DelFly case. This is mostly because of the varying dynamics of the helicopter. The fact that the winner of the competitions used high-level methods provides a strong indication that the DelFly may also require such methods. However, there are a number of reasons why the analogy between the DelFly and the RL Competitions is not perfect:

- A helicopter in hover is typically unstable, and requires advanced nonlinear controllers for stability. The DelFly is stable in most flight conditions, and can be controlled with simple cascaded PID controllers [10]. This makes it harder to find safe controllers for the helicopter task.
- Optimizing the advanced helicopter neurocontrollers with many parameters is more difficult than optimizing controllers with few parameters, which may be suitable for the DelFly.
- In the RL Competitions, only the wind varies. Although this was unknown to the participants, this task is less general than the control of the DelFly, where everything in the dynamics may vary. Koppejan

[5] http://www.rl-competition.org/.

[6] The robustness of these controllers was evaluated among the training models.

and Whiteson [24] present a more general hovering task, and propose a very different control method[7] for this task. This is an indication that the hybrid selection approach may not be suitable for fully generalized tasks.

- The training examples for the helicopter task are drawn from the same distribution as the test cases. For the DelFly, the aerodynamic model structure is not matched to the actual dynamics. This means that the variation between the identified models is not necessarily equal to the variation between actual DelFly's.

- The episodes in the helicopter hovering task take ten minutes, which helps to counteract noise in the policy evaluation. Such episodes are clearly too long for the DelFly. It is unknown whether the noise and disturbance levels for the DelFly are low enough to allow policy evaluation within a suitable time.

It is concluded that the conclusions from the helicopter hovering competitions cannot be directly generalized to the DelFly. Nevertheless, the idea of high-level selection of predefined controllers seems very suitable, and should be considered for the DelFly.

7.5.2. Q-learning for local controller selection

In the previous subsection, it was proposed to select one of several predefined controllers. This subsection considers a local variant of this approach.

Molenkamp et al. [32] investigated the use of two NDI controllers for a quadrotor. The state space is discretized, and the action space consists of two possible actions, corresponding to the two NDI controllers. Q-learning is used to find the best NDI controller for every state.

Instead of using one complicated global controller, which is either adaptive or selected from a set of predefined controllers, it is possible to define multiple simple local controllers. The RL algorithm can then select the most suitable controller in any flight state. This approach may be very effective in tackling the issue of local models for the DelFly. The key difference with the goals in [32] is that for the DelFly, this method may also help against the variability between individual DelFly's.

As an example, consider two PID controllers (1 and 2), and two different DelFly's (A and B). It is possible that controller 1 provides good behavior for the whole flight envelope of DelFly A. DelFly B is different, however: it requires controller 1 for low speeds and controller 2 for high speeds. When applied in this setting, the method in [32] becomes a local variant of the approach in the previous subsection. The advantage is that the local controllers can be simple and linear. After learning, the resulting controller is similar to a linear controller with gain scheduling. However, the learning process takes considerably longer than for the previous algorithm. This approach is therefore discarded.

7.5.3. Optimistic model predictive control

A new approach to the 2008 helicopter hovering competition was proposed by [31]. The method learns a stochastic Dirichlet Process Mixture Model (DPMM) of the environment. This model is optimistic when uncertainty is present. Model Predictive Control (MPC) is used to optimize the control policy. This method uses only an actor, but does not make use of experienced reward. It therefore does not fall directly in any of the categories. The algorithm is able to obtain hovering behavior after 5 ± 2 seconds [31]. Considering that an episode in the hovering task takes 600 s, this is a huge improvement over the previous results. This method is disregarded because of its computational requirements. It is mentioned that every second of helicopter simulation requires 100 s of computation for the control algorithm [31]. Although this is highly dependent on parameters like model complexity, sample rate and computational power of the processor, it appears that online implementation on the micro-controller of the DelFly will not be feasible in the near future. Nevertheless, this direction of research should be closely monitored. This approach is one of the main inspirations for the newly developed CAML algorithm.

7.6. Preselection from literature

Many RL controller options were discussed in the previous sections. It was found that the actor-critic algorithms have been compared reasonably well in literature. Algorithms within the critic-only, actor-only and high-level classes, however, have hardly been compared directly. The amount of comparison between the

[7] This method is considered to require too many samples, and will not be discussed further.

different classes is limited as well. Without any particular order, the most promising algorithms are listed below:

- HDP
- J-SNAC
- Policy gradient with inaccurate models
- High-level controller selection

It is noted that the choice of J-SNAC over SNAC is not based on direct comparison. The claims by the authors and the analogy to DHP/HDP are used to predict that J-SNAC will learn faster.

It is difficult to compare the different algorithms by means of a literature study, because the performance of the different classes is expected to be highly problem-specific. Nevertheless, some general conclusions can be drawn. It was found that fast convergence is obtained with the policy gradient algorithm in [1], within sixteen episodes for the linearized F-16 aircraft [21]. Van Kampen et al. [46] analyzed the adaptability of an HDP controller on the F-16 aircraft, and found convergence within seconds. This is much faster, but the model change in the latter study is limited to one aerodynamic coefficient, whereas the policy gradient study changed all components of the model.

Optimal feedback control theory results in the Linear Quadratic Regulator (LQR) for linear systems, which requires full state feedback. However, some application studies (e.g. [46]) have attempted to simplify the structure by only using certain states. For actor-critic and actor-only controllers, it is also possible to use a special controller structure [18, 24, 46], which adds expert knowledge to the controller. It is expected that the controller structure has a large effect on learning speed. The same can be said about the function approximators: while feedforward neural networks benefit from their universal approximation capabilities, the nonlinear backpropagation process may take much longer than required with simple linear structures.

8

Implementation and training of intelligent agents

The previous chapter introduced several Reinforcement Learning solutions for flight control. The outlines of the several methods were sketched, and differences were indicated. Since the algorithms share some implementation details, these will be considered in this chapter. Section 8.1 discusses the use of neural networks as function approximators. The distinction between several solution schemes is discussed in Section 8.2.

8.1. Neural networks

Neural Networks (NNs) are inspired by the nervous system of living creatures. They consist of multiple layers, connected by weights. A simple form of the neural network, the Multi-Layer Perceptron (MLP), is shown in Figure 8.1.

The MLP in Figure 8.1 is said to have one input layer, one output layer and one hidden layer, consisting of the $\sum$ and σ operators. The elements are called neurons: there are two input neurons, two output neurons and three hidden neurons. The hidden neurons consist of the sum and σ operators. Often, only the σ is drawn, and the summation is implicit. Every input u_i is fed to every sum operator. Additionally, a bias is connected as an additional input. All these connections are weighted by scalars $w^1_{j,i}$. For each hidden neuron, they are summed to result in f_j. This is shown in Eq. 8.1. This relation can also be written in matrix form, if the bias is considered an additional input ($\bar{u} = [u^T 1]^T$). This is done by placing the weights at the position in the weight matrix $\bar{W}^1$ corresponding to their indices. The resulting relation is shown in Eq. 8.2. The notation is adapted from [17] and [46].

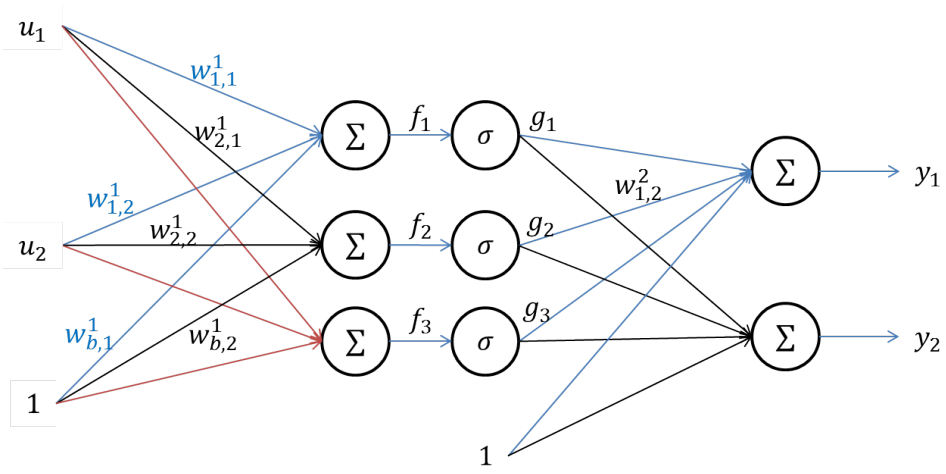


Figure 8.1: Schematic representation of the Multi-Layer Perceptron. Some weights are named to illustrate the numbering convention.

$$f_j = \sum_{i=1}^{m} \left[w^1_{j,i} u_i \right] + w^1_{b,j} \tag{8.1}$$

$$f = \bar{W}^1 \bar{u} \tag{8.2}$$

The f_j are processed separately by the σ operators. These activation functions, which are usually but not necessarily equal for all neurons, perform a nonlinear function on their inputs. These are often sigmoid functions. In ACDs, this is commonly the scaled exponential form of the hyperbolic tangent [42, 46], shown in Eq. 8.3. Another example the logistic function in Eq. 8.4. The derivatives of these functions are also shown. LeCun et al. [25] argue that the hyperbolic tangent function is better, because it is symmetric about the origin.

$$\sigma(f) = \frac{1 - e^{-f}}{1 + e^{-f}} \qquad \frac{d\sigma(f)}{df} = \sigma' = \frac{1}{2}\left(1 - f^2\right) \tag{8.3}$$

$$\sigma(f) = \tanh(f) \qquad \frac{d\sigma(f)}{df} = \sigma' = 1 - f^2 \tag{8.4}$$

$$\sigma(f) = \frac{1}{1 + e^{-f}} \qquad \frac{d\sigma(f)}{df} = \sigma' = f(1 - f) \tag{8.5}$$

The step from the hidden layer to the output layer is very similar to the step from the input layer to the hidden layer. This is shown in Eq. 8.6 and 8.7, with $\bar{g} = [g^T 1]^T$ [17, 46].

$$y_k = \sum_{i=1}^{m} \left[w^2_{k,j} g_j \right] + w^2_{b,k} \tag{8.6}$$

$$y = \bar{W}^2 \bar{g} \tag{8.7}$$

The output of the network is a linear sum of the outputs of the hidden neurons (and the bias). For actor networks, an option is to replace the linear output neurons by appropriately scaled sigmoids [46]. With this method, actuator constraints can be forced onto the actor output.

If a sufficient amount of hidden neurons is available, the MLP with a single hidden layer of sigmoids can approximate any continuous function in the unit hypercube [9]. The output bias is not strictly required for this. MLPs with multiple hidden layers are referred to as deep neural networks.

Neural networks are often trained by means of *backpropagation* [25]. This is a gradient descent algorithm, which makes use of the explicit knowledge of the sigmoid function. The key is to determine the gradient of some cost function with respect to the parameters. If a cost function $C = h(y)$ is known, one can easily determine $\frac{\partial C}{\partial y}$ as the gradient with respect to the outputs.

First, consider the case with one output. The output relation can then be written as $y = w^T \bar{g}$, with a column vector w. Note that this column vector corresponds to a row in the matrix $\bar{W}^2$. With this notation, the gradient with respect to the output weights is determined according to Eq. 8.8 [17, 46].

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial y} \frac{\partial y}{\partial w} = \frac{\partial C}{\partial y} \bar{g}^T \tag{8.8}$$

If multiple outputs are present, the weights form a matrix. The chain rule cannot be applied directly in this case, since it would require the derivative of vector y to matrix $\bar{W}^2$, which is not well-defined. This is solved by writing the individual outputs $y_k = \left(w^2_k\right)^T \bar{g}$, and using Eq. 8.9 [17, 46]. This can be done because the outputs do not depend on all weights in W^2; they only depend on the weights in one row.

$$\frac{\partial C}{\partial w^2_k} = \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial w^2_k} = \frac{\partial C}{\partial y_k} \bar{g}^T \tag{8.9}$$

This gradient can be used to update the output weights, along with the output bias. The input weights are determined by backpropagation through the hidden neurons. First, the derivatives with respect to the vector g are determined using Eq. 8.10 [17, 46]. Since the partial derivative to the output bias is unimportant, the last component of this (row) vector can be discarded to arrive at $\frac{\partial C}{\partial g}$. The gradient with respect to the hidden neuron input f is then calculated according to Eq. 8.11 [17, 46].

$$\frac{\partial C}{\partial \bar{g}} = \sum_k \frac{\partial C}{\partial y_k} \frac{\partial y_k}{\partial \bar{g}} = \sum_k \frac{\partial C}{\partial y_k} \frac{\partial \left(w_k^2\right)^T \bar{g}}{\partial \bar{g}} = \sum_k \frac{\partial C}{\partial y_k} \left(w_k^2\right)^T \tag{8.10}$$

$$\frac{\partial C}{\partial f} = \frac{\partial C}{\partial g} \frac{\partial g}{\partial f} \tag{8.11}$$

Backpropagation through the hidden neurons occurs by using the analytical expressions for the derivatives of the activation functions in one of the Eqs. 8.3 to 8.5. Since every component of g only depends on the corresponding component of f, $\frac{\partial g}{\partial f}$ will be a diagonal matrix. The next step is to determine the derivatives with respect to the input weights. This is done in the same way as in Eq. 8.9, by splitting up the matrix W^1 into vectors w_j^1. This results in Eq. 8.12 [17, 46]. If desired, the gradient with respect to the inputs can be calculated according to 8.13 [17, 46].

$$\frac{\partial C}{\partial w_j^1} = \frac{\partial C}{\partial f_j} \frac{\partial f_j}{\partial w_j^1} = \frac{\partial C}{\partial f_j} \bar{u}^T \tag{8.12}$$

$$\frac{\partial C}{\partial \bar{u}} = = \sum_j \frac{\partial C}{\partial f_j} \frac{\partial f_j}{\partial \bar{u}} = \sum_j \frac{\partial C}{\partial f_j} \frac{\partial \left(w_j^1\right)^T \bar{u}}{\partial \bar{u}} = \sum_j \frac{\partial C}{\partial f_j} \left(w_j^1\right)^T \tag{8.13}$$

For deeper NNs, this procedure can be continued for every layer. If the gradients to all weights are known, the weights are updated in the direction of the gradient to minimize J. This can be done according to Eq. 8.14, for every layer transition a with the receiving neurons distinguished by b.

$$\Delta w_b^a = -l(t) \left[\frac{\partial C}{\partial w_b^a} \right]^T \tag{8.14}$$

Backpropagation can be performed every time a new sample becomes available. This method is called *stochastic learning* [25]. The other end of the spectrum is called *batch learning* [25], where the sum (or average) of reward over all samples is maximized. LeCun et al. [25] claim that stochastic learning has the advantage of being faster. It is also said to lead to better performance, because the noise in the estimation of the gradient leads to exploration. A third advantage is that it can be used to track changes [25]. The advantages of batch learning are due to a better understanding of the theory, which comes with some more advanced techniques [25]. However, Du and Swamy [17] report faster learning and a higher accuracy for batch learning in a case study considering classification. Mini-batch learning uses a small batch of stochastic samples, and can be considered a mixture of the previous two methods. Some practical tricks for training neural networks are given below [25]:

- The training data should be shuffled to learn faster. This is because similar sequential examples do not provide different gradients to the NN. Unfortunately, this is very difficult in online applications.
- The inputs should be normalized appropriately to have near-zero averages and similar variances.
- The weights should be initialized appropriately.
- The learning rates should be selected correctly. They should preferably not be the same for the different neurons. Learning rates for the input weights are typically larger than those for the output weights.
- The learning rates can be varying.

With Reinforcement Learning, special care should be taken to maximize data efficiency. This can be done, for example, by performing stochastic learning during each episode, combined with batch learning on the data after each episode.

8.2. Solution schemes

The actor-critic and critic-only methods in Chapter 7 are aimed at solving the Bellman equations. It was mentioned that the temporal difference schemes can be implemented in several ways. This section discusses some implementations. The Bellman equation is restated in Eq. 8.15.

$$J(t) = \gamma J(t+1) + U(t) \tag{8.15}$$

Wang et al. [49] highlight that this equation can be solved iteratively in two ways:

- Take $J(t+1)$ as constant, and update $J(t)$ towards $\gamma J(t+1)+U(t)$. This is the forward-in-time approach.
- Take $J(t)$ as constant, and update $J(t+1)$ towards $\frac{1}{\gamma}(J(t)-U(t))$. This is the backward-in-time approach.

The first approach is used in discrete dynamic programming. In both cases, an issue is to obtain $J(t+1)$. With the first approach, this is often done by means of a model if that is available. An alternative approach is to wait one time step, and to solve the Bellman Eq. 8.16.

$$J(t-1) = \gamma J(t) + U(t-1) \tag{8.16}$$

This makes it possible to solve the critic equation without a model. The critic weights at time $t-1$ are used to determine the error in the equation. In the forward-in-time approach, the right hand side of Eq. 8.16 is fixed (with the weights at time $t-1$), and $J(t-1)$ is moved by adapting the critic weights. This results in the weights at time t. The value $J(t)$ must be recomputed for the next time step, in order to avoid using old weights. If the value function is action dependent, the actor must also be recomputed.

This recomputation makes it more logical to apply backward-in-time approaches if a model is not available. This was done in the model-free context of ADHDP [18, 42]. The backward-in-time approach may pose convergence issues. In preliminary research, this approach only converged with a negative learning rate. Literature [18, 42] reports positive learning rates, however.

The backward-in-time approach is most useful for ADHDP, in order to allow model-free control. In this research, HDP will be applied with a forward-in-time approach. This is done because the model is required anyways; there seems to be no reason to wait one time step. Also, this results in a fair comparison with J-SNAC, which also uses a model-based forward-in-time approach.

In all cases that were discussed, the states $x(t)$ and (a prediction of) $x(t+1)$ are available. One may wonder why it is not more logical to vary both $J(t)$ and $J(t+1)$ in order to bring the critic error to zero. Werbos [51] performed a consistency analysis of HDP on a simple system. He concluded that if disturbances are present, moving both values with gradient descent gives an incorrect result. The reader is referred to Werbos [51] for a detailed derivation. An intuitive explanation is provided in this section. When minimizing the squared critic error, the gradient direction is given in Eq. 8.17.

$$\frac{\partial E_c(t)}{\partial \theta_i} = \frac{\partial E_c(t)}{\partial e_c(t)} \frac{\partial e_c(t)}{\partial \theta_i} = e_c(t) \frac{\partial e_c(t)}{\partial \theta_i} \tag{8.17}$$

The term $e_c(t)$ contains $J(t+1)$, and is therefore dependent on the disturbance. If $J(t+1)$ is not held fixed, $\frac{\partial J(t+1)}{\partial \theta_i} \neq 0$. This means that $\frac{\partial e_c(t)}{\partial \theta_i}$ will also depend on the disturbance, and a quadratic disturbance term appears in Eq. 8.17! This term does not go to zero in expectation, resulting in a bias in the gradient. Werbos [51] found that this results in incorrect value functions for a simple system if both $J(t)$ and $J(t+1)$ are moved, while the forward-in-time approach gives the correct result. By the reasoning in this section, it seems that the backward-in-time approach will also give incorrect results.

The goal of this iterative update rule is to minimize the squared TD error in the critic. If the function approximator is linear in its parameters, it is also possible to use least-squares methods to minimize this error for multiple time steps at once.

Least-squares methods have been widely used to update critic weights in Reinforcement Learning [29]. An in-depth review of least-squares methods for discrete MDPs is provided by Buşoniu et al. [4]. The elaboration in this report is based on the approach in [29]. The critic error is restated in Eq. 8.18. If the function approximator for the critic is linear in its parameters θ, it can be written as in Eq. 8.19 , where $\phi(x,u)$ denotes a column vector of basis functions. The critic error can then be written as in Eq. 8.20 [29].

$$e_c(t) = J(x(t), u(t)) - \left[\gamma J(x(t+1), u(t+1)) + U(x(t), u(t))\right] \tag{8.18}$$

$$J(x, u) = \phi^T(x, u)\theta \tag{8.19}$$

$$e_c(t) = \left[\phi(x(t), u(t)) - \gamma\phi(x(t+1), u(t+1))\right]^T \theta - U(x(t), u(t)) \tag{8.20}$$

Equation 8.20 represents the critic error at one time step. Least-squares can be used to minimize this error over several time steps. This is done according to Eqs. 8.21 to 8.23 [29], where the dependence of $\phi(t)$ and $U(t)$ on the state and input is implicit.

$$A = \begin{bmatrix} \left[\phi(0) - \gamma\phi(1)\right]^T \\ \left[\phi(1) - \gamma\phi(2)\right]^T \\ \vdots \\ \left[\phi(N-1) - \gamma\phi(N)\right]^T \end{bmatrix} \tag{8.21}$$

$$B = \begin{bmatrix} U(0) & U(1) & \dots & U(N-1) \end{bmatrix}^T \tag{8.22}$$

$$\theta = \left[A^T A\right]^{-1} A^T B \tag{8.23}$$

This provides the optimal solution for θ over the whole data set. It is therefore not bothered by the forgetting issues of backpropagation. The least-squares process may also be written in a recursive form using the Recursive Least Squares (RLS) algorithm [22]. If the dynamics of the system are time-varying, it is possible to introduce a forgetting factor.

9

Algorithm preselection

After the literature survey, several algorithms remained. It was explained in Chapter 2.4 that several intermediate tests were performed before the final test problem. This chapter focuses on these tests. Section 9.1, 9.2 and 9.3 describe the three problems in the preliminary test phase. The *Step* test is elaborated in Section 9.4.

9.1. Complex dynamics

The first preliminary test problem is aimed at finding the required complexity of the controller structure. This problem directly compares HDP and J-SNAC.

9.1.1. Problem description

The system to be controlled in this test problem is a linearized DelFly model. A model with trimmed condition at 0.6 m/s was taken for this. The model is rewritten to use the horizontal speed in the earth-fixed reference frame as a state, rather than the axial velocity. The goal of the controller is to regulate the DelFly to its trimmed speed state (which is zero due to the use of a linearized model). The reward function in Eq. 9.1 punishes the squared deviation from this state, as well as the squared control deflection. There is no penalty for divergence. This test problem is deterministic: measurement noise and disturbances are not present. The problem is undiscounted.

$$U = -Qv^2 - Ru^2 \tag{9.1}$$

The strength of this problem is that it the optimal control solution is known, and given by the Linear Quadratic Regulator (LQR). The LQR solution results in the optimal feedback control $u = Kx$, where linear full-state feedback is used. Also, the optimum value function is available. This is a quadratic function of the full state ($J = x^T Px$). Although the reward function depends on only one of the states, the value function is dependent on each of the states. The reward function is selected such that the value function is close to one on average.

It is known that gradient descent algorithms perform much better if the states are in the same order of magnitude. Because of this, the states are divided by a reference value of $x_{ref} = \begin{bmatrix} 0.36 & 0.09 & 0.10 & 0.07 \end{bmatrix}^T$. These values were estimated by looking at simulated model responses to a 3211 input. Because of this, the ratios between the reference states are motivated, but their absolute values are determined only by the magnitude of the control deflection during the simulation. When using the normalized state, the matrices of the optimal feedback and value function are given by Eq. 9.2.

$$K = \begin{bmatrix} -0.999 & 0.199 & -0.0354 & -1.78 \end{bmatrix} \qquad P = \begin{bmatrix} -0.0377 & -0.0129 & -0.00262 & -0.0693 \\ -0.0129 & -0.0962 & -0.00632 & -0.0331 \\ -0.00262 & -0.00632 & -0.00224 & -0.00608 \\ -0.0693 & -0.0331 & -0.00608 & -0.129 \end{bmatrix} \tag{9.2}$$

The value function is dominated by V and θ. The optimal control input is influenced most by q and θ. It can be seen that w hardly affects any of these functions. For this reason, it was decided to develop a controller based on q, V and θ.

J-SNAC has only been flight-proven with full state feedback. It is therefore interesting to investigate whether it is possible to use this algorithm for control of partially modeled dynamics.

Theoretically, the value function (implemented by the critic) is a quadratic form, and the optimal feedback (implemented by the actor) is linear. In the simplest implementation considered, this knowledge is used by using a second order polynomial for the critic, and a linear feedback matrix for the actor. Also, feedforward Neural Networks (NNs) are attempted with backpropagation. Ten hidden neurons with sigmoid activation function are used.

The models in this test problem are linear black-box models. Just like for the actor and critic, only three states are used for the models. They are identified online using either backpropagation or recursive least squares. The updating method for the model always matches the updating method for the critic.

With HDP, it is common to pre-train the different parts of the controller separately, before starting Reinforcement Learning. However, it was found that large 'shocks' still occur once Reinforcement Learning is started. In this test problem, it is instead attempted to initialize the critic's (output) weights at zero. This makes sure that the actor initially does not learn (because the derivatives of the critic are zero as well). The model weights are initialized randomly, and are typically learned more rapidly than the critic and actor weights. The actor's weights are also initialized randomly. For the linear actor, however, instability is often obtained for random weights. The linear actor's weights are therefore initialized randomly between 0 and 2 times the LQR optimal weights. When RLS is used, the parameter covariance matrices are initialized with 10000 on all diagonal elements, and 0 on all off-diagonal elements.

The performance measure during this test problem is the cumulative reward obtained during the test. This is divided by the number of episodes to result in the average cumulative reward per episode. Although multiple episodes are considered, the task is not episodic; the DelFly is only reset to a new initial state when certain conditions are met. On this resetting step, the controller is not updated.

With random initial states, it was found that the initial state has a much larger effect on the reward than the controller settings. Therefore, a large number of trials is required to find the effect of the controller settings. This is circumvented by using the optimal value function from the LQR. The controller is always started at a random state with $J = -1.1$, where care is taken to avoid excessively large states. This makes it possible to compare the performance of the RL controllers to an optimum. Also, it adds some regularity to the task, which may be exploited by the Neural Network-based controllers.

9.1.2. Results

With J-SNAC, it is found that the initialization is very important. No converging runs were found when using NNs. With the linear-quadratic function approximators, it was found that RLS leads to large sudden changes in the critic, which are amplified to result in even larger changes in the selected action. With backpropagation, seven out of ten runs diverge.

A problem occurs when the model used for J-SNAC is learned online. If the actor is linear, the control deflection is a linear function of the states. This makes it impossible for the model to distinguish between the influence of the states and the influence of the control deflections. It was expected that the variation in the actor weights would solve this problem, but it is found that the learning often gets stuck in a suboptimal point. Adding exploratory action selection does not help sufficiently. Literature [14] recommends a different model updating method with Lyapunov stability, but this was not attempted. Another limitation of this approach is that stochastic gradient descent was used with the backpropagation algorithms, while the original J-SNAC publication [15] recommends a special batch updating method (which is only suitable for offline learning). Informal analysis with mini-batch updating did not yield improved results.

HDP was implemented with both types of function approximators. The polynomial critic learns most rapidly. However, poor convergence is found with backpropagation. This is expected to be due to the crude parameter settings, which are all powers of ten. With RLS, rapid convergence to a global optimum is found for all runs considered.

With NNs as function approximators, converging behavior was found. However, learning is slower than with linear-quadratic function approximators. The relation between the learning rates of the actor, critic and model is very important for convergence. This seems to be more important than the magnitudes of these learning rates. It is expected that more accurate hyperparameter tuning may improve performance. With the best settings, all runs converge to a local minimum, but none of them finds the global optimum.

9.2. Variability

The goal of the *Variability* test problem is to assess the adaptability of the RL controllers. Two DelFly's are used for this problem: One model trimmed at 0.6 m/s (DelFly A) and one model trimmed at 1 m/s (DelFly B).

9.2.1. Problem description

A regulator problem is performed on the two linearized models. The controller is first trained for DelFly A, until the optimum is found. Then, it is used as an initial setting for DelFly B. The controller task is exactly the same as for the *Complex dynamics* test.

The same combinations of algorithms, function approximators and update method as during the *Complex dynamics* test were attempted. Even though certain combinations did not perform well, it is possible that the smaller changes and better initialization during the *Variability* test makes some combinations feasible.

When using RLS, the parameter covariance matrix reflects the confidence in the current critic/model. It was previously initialized at very high values to reflect the limited confidence. In this test problem, the initialization is considered a variable in the optimization process. The same value is used on all diagonal elements of the critic and model.

The training of the actor an critic for DelFly A can be skipped by using the LQR solution. The optimal actions and values are obtained from the LQR solution, and the actor and critic are trained towards these values in a supervised setting. For the linear-quadratic function approximator, the correct values can be set directly.

9.2.2. Results

In the previous test, it was found that HDP with polynomials and backpropagation diverges in most of the cases. This test does not show divergence for any of the ten trials (with the best parameter setting). However, only part of the controllers converges to a global optimum; the others are still drifting around without clearly converging or diverging. RLS performs better; with the best initialization of the parameter covariance matrices, all runs converge to the global optimum.

The pre-trained NN actor and critic are based on the LQR solution, and therefore look like a linear actor and quadratic critic. It is found that the controller can learn effectively within 500 episodes, but the optimal control solution is not reached. This number of episodes is clearly too large for any practical application, but it may be possible to lower this number by performing simulated episodes as well.

J-SNAC is also attempted for the *Variability* problem. With backpropagation, converging runs were again not obtained. J-SNAC with RLS is very sensitive to parameter settings: most settings result in ten diverging runs, but some result in ten converging runs. Even during these runs, performance is not as good as with HDP. It is therefore concluded that HDP is more suitable for the current application.

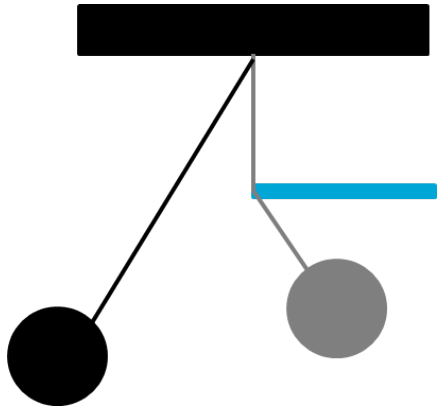
Some additional tests were performed on this test problem. HDP was attempted with linear-quadratic function approximators that make use of all the four states, using backpropagation. Even though the fourth state is hardly used by the actor and critic, this results in much faster learning: the global optimum is found within 200 episodes. This may be because the perfect solution can now be formed by the actor and critic. It was attempted to add some noise to the measurements, to avoid finding this perfect solution. Even with a Gaussian noise with a standard deviation of 5% of the RMS values of the states, performance hardly degrades. It is concluded from this that the HDP algorithm is sensitive to unmodeled dynamics, and should preferably be used with full state feedback.

J-SNAC was also attempted with full state feedback. Adding the fourth state improves performance over the three-state version of the algorithm. However, two out of ten runs diverge now. The performance of the four-state J-SNAC algorithm is still not as good as the performance of the four-state HDP algorithm. Furthermore, the performance greatly decreases when noise is added, and six out of ten runs diverge in that case.

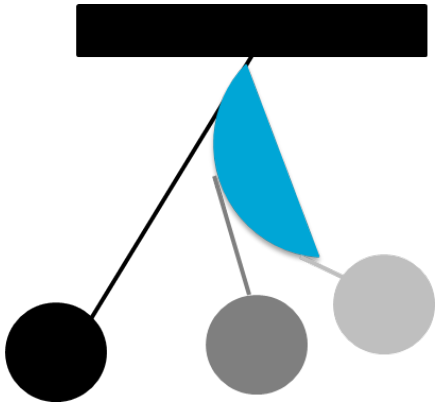
9.3. Local models

The available simulation models of the DelFly are all local linear models, which have been validated only for specific conditions. A challenge of this project is that a global controller should be trained for these models. This can be done in several ways:

- A global simulation model can be formed from the local models, and Reinforcement Learning is performed on this global model. A problem is that these global models have not been validated.



(a) Setup of the training pendulum. The length is shorter when the pendulum swings right.



(b) Setup of the testing pendulum. The length varies with deflection.

Figure 9.1: Systems used during the *Local models* test.

- Several local controllers can be trained for the local models, which are connected to form a global controller.

In both cases the training model will have a structure that does not necessarily match the underlying dynamics of the DelFly. The goal of the *Local models* test is to analyze which training method is more suitable.

9.3.1. Problem description

This test is performed on a simpler model than the DelFly. The model of interest describes the motion of a non-inverted pendulum. The motion is linearized to simplify the analysis. The pendulum is lightly damped, and a control actuator can apply force to the mass. The task of the controller is to regulate the deflection to zero. The reward function punishes the squares of the deflection and the input force. The problem is undiscounted. Even though the pendulum is naturally stable, the controller can improve the damping considerably.

The interesting part of the pendulum is its asymmetry: when it swings left, it behaves differently than when it swings right. The pendulum that is used for training has a length of 1 m during left deflections, and 0.5 m during right deflections. A graphical depiction is shown in Figure 9.1a.

HDP with NNs for the actor, critic and model is used for this problem. Applying this to the system directly is possible, but often results in local optima. The LQR solutions can be used for pre-training of the controller. When separate local controllers are trained, the left and right LQR solutions provide the perfect target for the left and right controllers. Supervised learning can be used to obtain the perfect NN actor and critic. For pre-training of the global controller, supervised learning is used first, where the targets are provided by the LQR solutions on both sides. This results in NNs that try to approximate a discontinuity at zero deflection. The pre-trained global controller was trained further with RL on the training pendulum to smooth away this discontinuity.

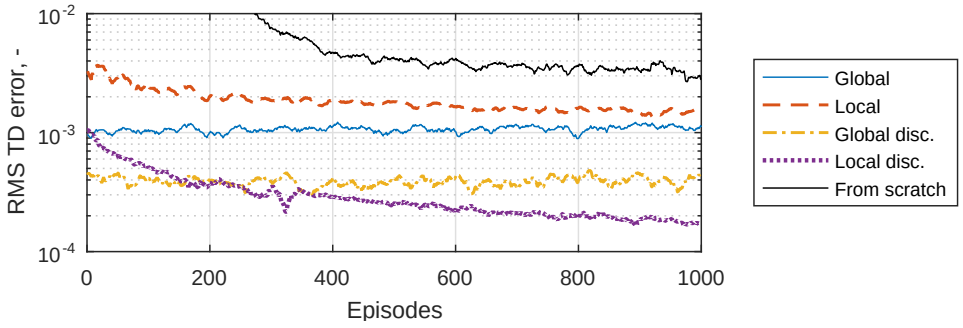
Just like in the *Variability* test, a different system is used to test the controller. The test pendulum has a linearly varying length, from 1 m at a -0.25 m deflection to 0.5 m at a 0.25 m deflection[1]. A graphical depiction[2] is shown in Figure 9.1b. Both controllers are assessed on their adaptability to this test pendulum.

9.3.2. Results

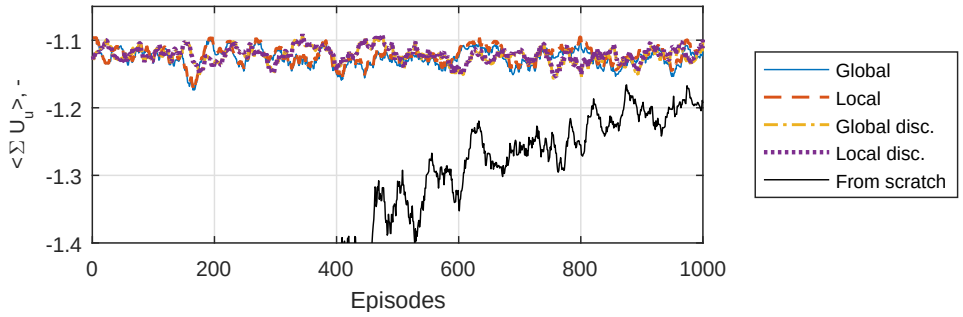
A first conclusion of the *Local models* test was obtained during pre-training. It was found that applying Reinforcement Learning from scratch was ineffective. This is rather unexpected, since the same algorithm did learn to control the DelFly during the previous two tests for a similar task. The asymmetric pendulum system has lower damping than the DelFly considered in the previous tests, which could explain some difficulty

[1] The linearization of the dynamics loses its validity at such large deflections. However, it is not intended to model a physical system accurately: the linearized problem remains interesting from a Reinforcement Learning point of view.

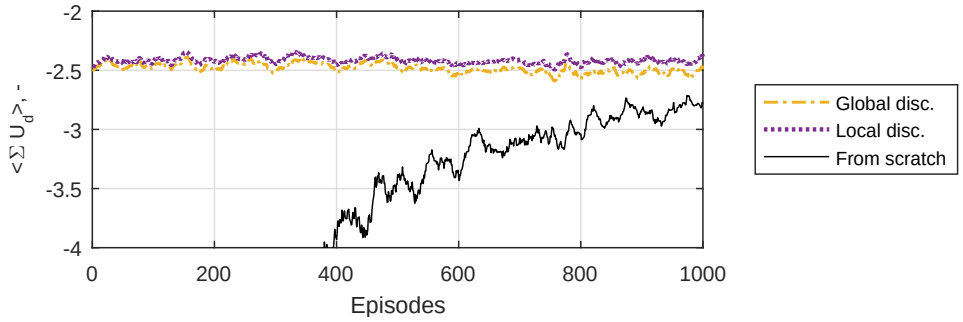
[2] It is acknowledged that the length of this pendulum system, if built in real life, would not vary linearly with deflection. This figure is intended for illustration purposes.



(a) Temporal Difference error.



(b) Mean cumulative reward per episode (equivalent undiscounted reward for discounted controllers).



(c) Mean cumulative discounted reward per episode.

Figure 9.2: Learning process of the undiscounted and discounted controllers on the *Local models* test. A discounted controller learning from scratch is shown for comparison.

in learning. However, performing Reinforcement Learning systems on unstable systems is very common in literature.

The local controllers are able to improve the performance of the critic over the initial values. The learning process is shown in Figure 9.2. For the best settings, all ten runs converge to a lower average Temporal Difference error. However, the actor does not seem to improve much, when judged by the mean cumulative utility per episode $< \sum U_u >$ in Figure 9.2b. No parameter setting was found for which the actor improves over time.

When testing the globally trained controller, it is found that much lower learning rates are required for the actor and critic to avoid divergence. Even when the learning rate is set 1000 times lower, good convergence is not obtained. It is found that the critic is actually diverging very slowly, just like the actor (the model learns as expected). Optimizing the learning rates of the input and output weights of the neural networks separately did not result in convergence[3].

9.3.3. Discounted learning

During preliminary research, it was found heuristically that the discount factor has a large effect on the learning speed of an HDP controller: lower discount factors result in faster learning. This problem has the highest discount factor of 1. Note that this problem is a regulator problem, and therefore has a final state, which will not be left once it is reached. With the current reward function, this state should clearly have a value of 0.

[3]Note that only powers of ten were considered for the learning rates. Finer optimization may improve performance.

This can be modeled as a one-state discrete Reinforcement Learning problem. Using Temporal Difference updating, which is also used via backpropagation in HDP, the value function is updated according to Eq. 9.3.

$$J_{i+1} = \gamma J_i + U \tag{9.3}$$

In this perfect state, $U = 0$. When the value function is initialized at some random value, it will asymptotically converge to zero if $|\gamma| < 1$. For an undiscounted problem, however, this value will remain constant! This reasoning can be extended to the continuous case. With HDP, the value function at the current state is updated towards the value of the next state plus the experienced reward, according to Eq. 9.4 (with α some carefully selected step size).

$$J(t) \leftarrow J(t) + \alpha(\gamma J(t+1) + U - J(t)) \tag{9.4}$$

Note that, if $\gamma = 1$, the Temporal Difference error does not change if both $J(t)$ and $J(t+1)$ are changed by the same constant. Therefore, the optimal value function has one degree of freedom: it can be changed by a constant. During the *Complex dynamics* and *Variability* tests, it was indeed found that the critic can change by a constant, without affecting the Temporal Difference error.

This also means that with $\gamma = 1$, the magnitude of the value function is irrelevant: it is only the difference between the values of two consecutive states that matters. The slope of the value function is therefore more important than the value function itself.

As a solution, it is proposed to use a discounted problem. It was found empirically that by redefining the reward function for a **given linear** system, a discounted problem with the same optimal control feedback law as the original undiscounted problem can be defined. This only works for a given system, but applying the new reward function to a similar system results in behavior that is close to optimal for the original reward function.

When using an equivalent discounted problem with $\gamma = 0.95$, the critic error can be reduced effectively in the *Local models* test, as shown in Figure 9.2a. It is found that the locally trained controller learns more rapidly than the globally trained controller. Both controllers can make use of the same learning rates now. When looking at the actor, however, improvements are again not found. Figure 9.2b shows the equivalent undiscounted reward, which does not improve. The mean cumulative discounted reward $< \sum U_d >$, which is directly fed to the controller, is depicted in Figure 9.2c; this also does not improve. It was also attempted to train a discounted controller from scratch, directly on the test pendulum (also included in Figure 9.2. It is found that this controller learns effectively, but does not surpass the performance of any of the previous systems. It is therefore concluded that the pre-trained controllers are already too close to optimal to result in any notable learning.

9.4. Step test

After the preliminary test phase, three algorithms remain: the Policy Gradient algorithm, CAML and HDP. These algorithms are implemented on a DelFly system, accounting for variability.

9.4.1. Problem description

The three algorithms are compared on an early first version of the final problem. This problem is similar to the final *Tracking* test, as described in Chapter 3, but has a number of differences.

- There is no pitch rate punishment: the reward function is only based on velocity.
- The 46 models used in this test are identified with Ordinary Least Squares (OLS), rather than Maximum Likelihood Estimation (MLE).
- All models are re-trimmed to the same trim point. This is done because the A-matrices are dependent on the trim settings. The aerodynamic coefficients are separated from the kinematics, and all models are placed at the same trim point (± 0.6 m/s) to avoid this issue.

The pitch rate punishment, which was added after this test, results in improved damping, as explained in Chapter 3. Using OLS rather than MLE seems like a small difference, but the OLS models are easier to control.

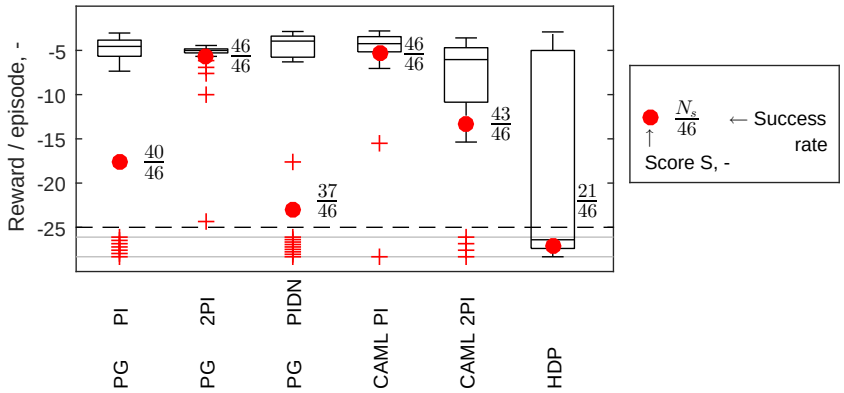


Figure 9.3: Results of an early test problem. These are not comparable to the final results.

This test uses the same algorithms as the *Tracking* test, but two additional options are used. First, this test contains a Policy Gradient PID controller. A derivative filter is installed to avoid problems with tracking the step, such that a PIDN controller is obtained. Several filter coefficients N were attempted. An attempt was made to design PIDN controllers for CAML as well. However, it was found that the offline optimization during the gain generation process results in widely varying gains if PIDN is used. Also, the derivative term results in lower safety[4]. This option was therefore discarded.

The second addition is that HDP was attempted on this problem. In agreement with the *Variability* test, all four states were used for the actor and critic, and the reference velocity was added as a fifth state. Taking into account the findings in Section 9.3, a discount factor of 0.9 was used. It was found that using Neural Networks results in much slower learning; linear-quadratic approximators are used instead. The actor is affine and has the structure of Eq. 9.5, with c the climb rate.

$$\delta_e = \begin{bmatrix} K_q & K_V & K_c & K_\theta & K_{ref} & K_0 \end{bmatrix} \begin{bmatrix} q & V & c & \theta & V_{ref} & 1 \end{bmatrix}^T \tag{9.5}$$

The critic is a polynomial of degree two, using all states (including the reference velocity). The model is linear. The parameters were initialized at the LQR solution of the average model. It was found that this often results in initially unstable controllers. A better actor initialization is found by trial and error (stabilizing 44/46 models), and the optimal value function with this actor is determined.

9.4.2. Results

These changes make the results of this test incomparable to the final results. This is why the results of this test are shown for all algorithms. Note that the algorithms used during this test are early versions of the final algorithms. The results are shown in Figure 9.3.

It can be seen that the PIDN controller results in accurate tracking, but has a lower success rate, which lowers the combined score. This is the reason why PIDN controllers were disregarded in the final test. It should be noted that this version of the PG algorithm did not explicitly include step size limits. It was found later that such limits enhance safety. A recommendation for future research is to test PIDN controllers with suitable step size limits.

The test was made easier for the HDP controller: noise was removed, and scores are averaged over 50 episodes, rather than 10. Still, it is found that HDP results in much lower safety: only 21 of the 46 simulations are successful. Several techniques are available to increase the safety: variable learning rates, step size limits, different (nonlinear) function approximators, independent training of actor and critic and batch learning. However, it is expected that this algorithm will never meet the safety levels achieved by the PG and CAML controllers. This is the reason to disregard HDP during the final test.

[4]These issue could possibly be solved by the pitch rate punishment and step size limit that were added later.

10

Concluding the project

The DelFly is a difficult vehicle to control because of its flapping dynamics and nonlinearity. Also, there is variability between different individual vehicles, and measuring the state is difficult due to noise and disturbances. This project has analyzed the performance of Machine Learning controllers for the DelFly.

From the vast amount of algorithms in Reinforcement Learning literature, successive refinements were made, until only the Policy Gradient algorithm remained. This approach was extended with a new variable learning rate technique in order to improve safety.

Drawing inspiration from several RL algorithms, CAML was proposed as a new method to tackle such problems. This approach identifies a model, which is used to select the most appropriate PI gain set with a Neural Network classifier. The selectable gain sets are predefined. This algorithm can change the gains rapidly during flight, which may improve safety.

The contribution of this project to the body of science is threefold. First, some Reinforcement Learning algorithms have been compared objectively in a flight control context. It was found that the recent J-SNAC algorithm cannot match the performance of the better known HDP. Although HDP may lead to the optimal control solution, its learning speed and safety are dramatically lower than with the PG algorithm.

The second contribution is that a step towards a new DelFly controller has been taken. This controller is not yet sufficiently validated; flight tests are required to do so. Yet, extensive simulations into the capabilities of the algorithms have been performed, including sensitivity analyses. The main limitation is that the current algorithms cannot deal with the flapping motion very well. Some suggestions are made to improve this behavior, which may be relevant for future research.

It was found that a 'universal' controller exists, which is able to stabilize all models that can be stabilized with the controller structures considered. The Machine Learning controllers, which operate on a higher level, are unable to improve the safety because of this. Nevertheless, the PG algorithm can improve the tracking accuracy, also taking into account damping. The low-level controllers considered in this research are relatively simple. For more effective control, more complex controller structures should be analyzed.

The third contribution is the development of a new algorithm: the Classification Algorithm for Machine Learning control. This method is intended to increase safety by allowing rapid gain changes. Because some of the models used in this research could not be controlled by the underlying low-level controllers, and the others were stabilized by the universal controller, this research was unable to prove this safety advantage of CAML. It is recommended to test CAML on a problem where a universal controller is not available. Also, one may include different controller structures in the selection possibilities. CAML is a method that can be applied to any system and task. Several suggestions for further improvement of this algorithm are given. For future research, it is recommended to demonstrate the capability of increasing safety first. Next, a more theoretical analysis is considered appropriate, such that the method can reach a state of higher maturity.

This project has contributed to both the solution of practical problems and the theory on Machine Learning control. Possibilities for further research lie in both areas. The ultimate goal of improving aviation safety still lies far ahead. One way to reach it, is to continue the search for knowledge in the field of flight control.

Bibliography

[1] P. Abbeel, M. Quigley, and A. Y. Ng. Using inaccurate models in reinforcement learning. In *International Conference on Machine Learning (ICML)*, June 2006.

[2] S. F. Armanini, C. C. de Visser, G. C. H. E. de Croon, and M. Mulder. Time-varying model identification of flapping-wing vehicle dynamics using flight data. *Journal of Guidance, Control, and Dynamics*, 39(3): 526–541, March 2016.

[3] S. F. Armanini, M. Karásek, C. C. de Visser, G. C. H. E. de Croon, and M. Mulder. Flight testing and preliminary analysis for global system identification of ornithopter dynamics using on-board and off-board data. In *AIAA Atmospheric Flight Mechanics Conference*, January 2017.

[4] L. Buşoniu, A. Lazaric, M. Ghavamzadeh, R. Munos, R. Babuška, and B. de Schutter. Least-squares methods for policy iteration. In M. Wiering and M. van Otterlo, editors, *Reinforcement learning: State-of-the-Art*, volume 12 of *Adaptation, Learning and Optimization*, chapter 3. Springer-Verlag Berlin Heidelberg, 2012.

[5] J. V. Caetano, C. C. de Visser, G. C. H. E. de Croon, B. D. W. Remes, C. de Wagter, J. Verboom, and M. Mulder. Linear aerodynamic model identification of a flapping wing MAV based on flight test data. *International Journal of Micro Air Vehicles*, 5(4):273–286, December 2013.

[6] S. Chen, Y. Yang, S. N. Balakrishnan, N. T. Nguyen, and K. Krishnakumar. SNAC convergence and use in adaptive autopilot design. In *International Joint Conference on Neural Networks*, pages 530–537, June 2009.

[7] T. Cunis. Precise position control of a flapping-wing micro air vehicle in a wind-tunnel. Master's thesis, RWTH Aachen University, May 2016.

[8] T. Cunis, M. Karásek, and G. C. H. E. de Croon. Precision position control of the DelFly II flapping-wing micro air vehicle in a wind-tunnel. In *International Micro Air Vehicle Conference and Competition (IMAV)*, October 2016.

[9] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, December 1989.

[10] G. C. H. E. de Croon, M. A. Groen, C. de Wagter, B. D. W. Remes, R. Ruijsink, and B. W. van Oudheusden. Design, aerodynamics and autonomy of the DelFly. *Bioinspiration & Biomimetics*, 7(2), May 2012.

[11] G. C. H. E. de Croon, M. Perçin, B. D. W. Remes, R. Ruijsink, and C. de Wagter. *The DelFly: Design, Aerodynamics and Artificial Intelligence of a Flapping Wing Robot*. Springer Netherlands, first edition, 2016.

[12] C. de Wagter, J. A. Koopmans, G. C. H. E. de Croon, B. D. W. Remes, and R. Ruijsink. Autonomous wind tunnel free-flight of a flapping wing MAV. In Q. P. Chu, J. A. Mulder, D. Choukroun, E. van Kampen, C.C. de Visser, and G. Looye, editors, *Advances in Aerospace Guidance, Navigation and Control: Selected Papers of the Second CEAS Specialist Conference on Guidance, Navigation and Control*, pages 603–621, April 2013.

[13] M. P. Deisenroth and C. E. Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *International Conference on Machine Learning (ICML)*, June-July 2011.

[14] J. Ding and S. N. Balakrishnan. Intelligent constrained optimal control of aerospace vehicles with model uncertainties. *Journal of Guidance, Control, and Dynamics*, 35(5):1582–1592, September-October 2012.

[15] J. Ding, S. N. Balakrishnan, and F. L. Lewis. A cost function based single network adaptive critic architecture for optimal control synthesis for a class of nonlinear systems. In *International Joint Conference on Neural Networks (IJCNN)*, July 2010.

[16] J. Ding, A. Heydari, and S. N. Balakrishnan. Single network adaptive critics networks - development, analysis, and applications. In F. L. Lewis and D. Liu, editors, *Reinforcement learning and approximate dynamic programming for feedback control*, pages 98–118. Wiley-IEEE Press, first edition, 2013.

[17] K. L. Du and M. N. S. Swamy. *Neural networks and statistical learning*. Springer London, 2014.

[18] R. Enns and J. Si. Helicopter trimming and tracking control using direct neural dynamic programming. *IEEE Transactions on Neural Networks*, 14(4):929–939, July 2003.

[19] S. Ferrari and R. F. Stengel. Online adaptive critic flight control. *Journal of Guidance, Control, and Dynamics*, 27(5):777–786, September-October 2004.

[20] S. Ha and K. Yamane. Reducing hardware experiments for model learning and policy optimization. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2620–2626, May 2015.

[21] J. L. Junell, T. Mannucci, Y. Zhou, and E. van Kampen. Self-tuning gains of a quadrotor using a simple model for policy gradient reinforcement learning. In *AIAA Guidance, Navigation, and Control Conference*, January 2016.

[22] V. Klein and E. A. Morelli. *Aircraft System Identification: Theory and Practice*. American Institute of Aeronautics and Astronautics (AIAA), Reston, Virginia, USA, 2006.

[23] J. A. Koopmans. Delfly freeflight. Master's thesis, Delft University of Technology, June 2012.

[24] R. Koppejan and S. Whiteson. Neuroevolutionary reinforcement learning for generalized control of simulated helicopters. *Evolutionary Intelligence*, 4(4):219–241, December 2011.

[25] Y. A. LeCun, L. Bottou, G. B. Orr, and K. R. Müller. Efficient backprop. In G. Montavon, G. B. Orr, and K. R. Müller, editors, *Neural networks: Tricks of the trade*, volume 7700 of *Lecture Notes in Computer Science*, chapter 1, pages 9–48. Springer Berlin Heidelberg, second edition, 2012.

[26] G. G. Lendaris. Adaptive dynamic programming approach to experience-based systems identification and control. *Neural Networks*, 22(5-6):822–832, July-August 2009.

[27] F. L. Lewis and D. Vrabie. Reinforcement learning and adaptive dynamic programming for feedback control. *IEEE Circuits and Systems Magazine*, 9(3):32–50, third quarter 2009.

[28] D. Liu. Neural network-based adaptive critic designs for self-learning control. In *International Conference on Neural Information Processing (ICONIP)*, November 2002.

[29] B. Luo, D. Liu, T. Huang, and D. Wang. Model-free optimal tracking control via critic-only Q-learning. *IEEE Transactions on Neural Networks and Learning Systems*, 27(10):2134–2144, October 2016.

[30] S. Lupashin, A. Schöllig, M. Sherback, and R. D'Andrea. A simple learning strategy for high-speed quadrocopter multi-flips. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1642–1648, May 2010.

[31] T. M. Moldovan, S. Levine, M. I. Jordan, and P. Abbeel. Optimism-driven exploration for nonlinear systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3239–3246, May 2015.

[32] D. Molenkamp, E. van Kampen, C. C. de Visser, and Q. P. Chu. Intelligent controller selection for aggressive quadrotor manoeuvring. In *AIAA Information Systems-AIAA Infotech @ Aerospace*, January 2017.

[33] M. Motamed and J. Yan. A reinforcement learning approach to lift generation in flapping MAVs: simulation results. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2150–2155, May 2006.

[34] M. Motamed and J. Yan. A reinforcement learning approach to lift generation in flapping MAVs: Experimental results. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 748–754, April 2007.

[35] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Autonomous inverted helicopter flight via reinforcement learning. In M. H. Ang and O. Khatib, editors, *Experimental Robotics IX*, volume 21 of *Springer Tracts in Advanced Robotics*, pages 363–372. Springer-Verlag Berling Heidelberg, 2006.

[36] B. D. Nichols. Continuous action-space reinforcement learning methods applied to the minimum-time swing-up of the acrobot. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 2084–2089, October 2015.

[37] G. J. Olsder, J. W. Van der Woude, J. G. Maks, and D. Jeltsema. *Mathematical Systems Theory*. VSSD, Delft, fourth edition, 2011.

[38] R. Padhi, N. Unnikrishnan, and S. N. Balakrishnan. Optimal control synthesis of a class of nonlinear systems using single network adaptive critics. In *American Control Conference*, pages 1592–1597, June-July 2004.

[39] R. Padhi, N. Unnikrishnan, X. Wang, and S. N. Balakrishnan. A single network adaptive critic (SNAC) architecture for optimal control synthesis for a class of nonlinear systems. *Neural Networks*, 19(10): 1648–1660, December 2006.

[40] D. V. Prokhorov and D. C. Wunsch II. Adaptive critic designs. *IEEE Transactions on Neural Networks*, 8 (5):997–1007, September 1997.

[41] J. W. Roberts, L. Moret, J. Zhang, and R. Tedrake. Motor learning at intermediate reynolds number: Experiments with policy gradient on the flapping flight of a rigid wing. In O. Sigaud and J. Peters, editors, *From Motor Learning to Interaction Learning in Robots*, volume 264 of *Studies in Computational Intelligence*, pages 293–309. Springer Berlin Heidelberg, 2010.

[42] J. Si and Y.T. Wang. Online learning control by association and reinforcement. *IEEE Transactions on Neural Networks*, 12(2):264–276, March 2001.

[43] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, first edition, 1998.

[44] Sjoerd Tijmons, Guido de Croon, Bart Remes, Christophe De Wagter, Rick Ruijsink, Erik-Jan van Kampen, and Qiping Chu. Stereo vision based obstacle avoidance on flapping wing mavs. In Q. P. Chu, J. A. Mulder, D. Choukroun, E. van Kampen, C.C. de Visser, and G. Looye, editors, *Advances in Aerospace Guidance, Navigation and Control*, pages 463–482, April 2013.

[45] H. van Hasselt. Reinforcement learning in continuous state and action spaces. In M. Wiering and M. van Otterlo, editors, *Reinforcement learning: State-of-the-Art*, volume 12 of *Adaptation, Learning and Optimization*, chapter 9. Springer-Verlag Berlin Heidelberg, 2012.

[46] E. van Kampen, Q. P. Chu, and J. A. Mulder. Continuous adaptive critic flight control aided with approximated plant dynamics. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, August 2006.

[47] G. K. Venayagamoorthy, R. G. Harley, and D. C. Wunsch II. Comparison of heuristic dynamic programming and dual heuristic programming adaptive critics for neurocontrol of a turbogenerator. *IEEE Transactions on Neural Networks*, 13(3):764–773, May 2002.

[48] J. L. Verboom, S. Tijmons, C. de Wagter, B. D. W. Remes, R. Babuška, and G. C. H. E. de Croon. Attitude and altitude estimation and control on board a flapping wing micro air vehicle. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5846–5851, May 2015.

[49] F. Y. Wang, H. Zhang, and D. Liu. Adaptive dynamic programming: An introduction. *IEEE Computational Intelligence Magazine*, 4(2):39–47, May 2009.

[50] J. Wang and J. Kim. Optimization of fish-like locomotion using hierarchical reinforcement learning. In *International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, October 2015.

[51] P. J. Werbos. Consistency of HDP applied to a simple reinforcement learning problem. *Neural networks*, 3(2):179–189, 1990.

[52] P. J. Werbos. Approximate dynamic programming for real-time control and neural modeling. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, fuzzy and adaptive approaches*, chapter 13. Van Nostrand Reinhold, 1992.

[53] P. J. Werbos. Neurocontrol and supervised learning: An overview and evaluation. In D. A. White and D. A. Sofge, editors, *Handbook of Intelligent Control: Neural, fuzzy and adaptive approaches*, chapter 3. Van Nostrand Reinhold, 1992.

[54] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, December 1945.