Faculty of Aerospace Engineering

Continuous state and action $\mathcal{Q}$-learning framework applied to quadrotor UAV control

A.E. Naruta

September 8, 2017

TU Delft Delft University of Technology

Challenge the future

Continuous state and action Q-learning framework applied to quadrotor UAV control

Master of Science Thesis

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

A.E. Naruta

September 8, 2017

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"Continuous state and action $\mathcal{Q}$-learning framework applied to quadrotor UAV control"** by **A.E. Naruta** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: September 8, 2017

Readers:

Dr ir E. van Kampen

T. Mannucci

Dr ir Q.P. Chu

Dr ir B.F. Santos

Contents

Chapter 1

Introduction

Reinforcement Learning(RL) is a field that has its roots in Artificial Intelligence(AI) community, which had emerged in the decades during and following the World War 2. RL methods aim to solve complex optimization and control tasks in an interactive manner. Reinforcement learning is a very attractive tool for identification and control of dynamical systems because it can deal with the nonlinearities that occur in most real-life systems. As such, it had been studied and applied to a wide range of problems in a variety of different fields. In reinforcement learning the system consists of an *agent*, acting in some *environment*. For example, an agent could be an autonomous vehicle, while the environment is the world in which it acts. The learning process itself consists of the agent taking different actions to *explore* the system and to *exploit* it by using the knowledge that it already has. If successful, the agent receives a reinforcement for its performance. This reward takes the shape of a certain *reward function*, determined beforehand. In autonomous vehicle context, the rewards could be defined by smooth movements, quick response, the safety of the operation, etc. The purpose of an agent is to maximize this reward by taking optimal actions. By acting in a certain way, the agent follows a *policy*, which represents the mapping from perceived states of the environment to actions to be taken (Sutton & Barto, 1998). While reward function determines the actions of an agent in a short run, influencing the policy taken, a *value function* is used to specify it's total reward over the long term.

A subset of reinforcement learning types is active learning. It describes the learning where learning systems not only passively collect and interpret data but in which they have at least some control over the inputs on which it trains. Active learning has advantages over purely passive learning that deals with randomly generated examples (Cohn, Atlas, & Ladner, 1994). The reasons for that is the quality of information that system input-output relations hold. For example, when exploring a given domain, it is more informative to survey the regions of higher uncertainty, to get a complete picture, as opposed to querying the area for which the information is already present. Although active learning has an established application domain in various fields of engineering, there is a lack of efficient methods designed to deal with nonlinear system identification based on automated probing (Bongard & Lipson, 2007), especially when applied to safety-critical physical processes.

Applications of RL in Unmanned Aerial Systems

Until recently the use of Unmanned Aerial Vehicles (UAV) was mostly limited to military applications. The technological advancements of the previous decade made the field of UAVs more accessible to private entrepreneurs and to the scientific community, which spurred an entirely new industry. Miniaturization of various electronics components combined with greater availability and lower prices contributed to this growing demand for UAVs. Right now UAVs are employed in numerous civilian applications, with new ones appearing every day.

While in the past only large multinationals such as Boeing, Airbus or Lockheed-Martin developed and sold sophisticated aerospace systems such as UAVs, lower technological barriers allowed numerous small companies to enter the field. A fully functional aerial platform could be designed and built in a short time span, using commercial off-the shelf components. It can be equipped with a sophisticated autopilot such as Paparazzi or ArduPilot, which in turn could be modified to perform any task or mission. And there is a continuously growing demand for these types of systems.

These developments, however, come with certain challenges. Operation safety of the UAVs constitutes one of the most paramount issues, particularly when UAVs operate in densely populated areas or interfere with air traffic. Since the field of commercial UAV operations is still relatively new, the legal framework that concerns the design, operation, and certification of new aerial vehicles is still in its nascent stages, and there are few guidelines regarding safety requirements. And in many countries, commercial operation of UAVs is not legal altogether, while in others it is strictly regulated, with high entry barriers for the operators. Another challenge is making these platforms more readily available to a wider audience of designers and engineers. With growing number of applications, developing new vehicles to fit a particular task still, requires experts who possess the necessary expertise and the know-how. This presents some difficulties when implementing a promising conceptual design into a real vehicle.

Recent interest in the field of unmanned vehicles had spawned numerous new applications for these platforms. As there are increasingly more ways to apply these vehicles, two key challenges arise: streamlining the development process while also making them safer.

The objective of this research is to develop an intelligent system capable of active learning for real-time control of a nonlinear plant. The intended application of such a system is to control a UAV. One of the key challenges in designing such a system is obtaining a desired degree of autonomy while taking safety issues into consideration. To achieve the stated objectives Reinforcement Learning methods can be applied.

Previous work

In the past couple of decades, there had been a growing interest in applying machine learning principles to aerial vehicles. Reinforcement learning is often applied to problems of adaptive control. In that case, if there is a change of plant configuration or environment configuration the controller can detect it and adapt accordingly. (Valasek, Tandale, & Rong, 2005),(Valasek, Doebbler, Tandale, & Meade, 2008) propose such an approach for control of Morphing Unmanned Air Vehicles. The policy in his approach consists of finding an optimal vehicle shape

change, given a particular flight condition. Structured Adaptive Model Inversion (SAMI) is used as a controller for tracking the trajectory, and a reinforcement learning module is used to produce the optimal shape. He uses two different approaches to reinforcement learning: Actor-Critic algorithm and Q-learning methods, which is a common temporal difference algorithm. The latter was applied to improve the accuracy of the algorithm. His approach used factors such as drag force and fuel consumption of the vehicle to calculate the cost of possible actions.

In 1994 NASA performed a 4-year research to demonstrate a concept for identification of aircraft stability and control derivatives using neural networks (Jorgensen, 1997). They have successfully demonstrated the capability of the system but also identified some issues with application of their method to real-time applications, e.g. high computational load that might compromise robustness of the system. In the following years, they did more research into the concepts, using it to develop an autoland system for the X-33 prototype aircraft (Cox, Stepniewski, Jorgensen, Saeks, & Lewis, 1999). In 2003 NASA published a technical memorandum titled Integration of Online Parameter Identification and Neural Network for In-Flight Adaptive Control(Hageman, Smith, & Stachowiak, 2003). There they have identified some of the advantages of learning systems over a non-learning system. Namely, they have observed that during the test maneuvers the system would immediately start to improve the response of the aircraft and a fully "learned" system showed excellent performance.

The concepts of reinforcement learning were also applied to rotorcraft control. (Bagnell & Schneider, 2001) designed and demonstrated an autonomous helicopter controller for the hover mode, designed using RL policy search methods. They validated their model using a Yamaha R50 helicopter and demonstrated the viability of their approach.

Apart from adaptive control, reinforcement learning is often used in risk-sensitive applications. The risk is used to denote constraints imposed upon parameters of the state-space. These are defined as forbidden or error states of the system, e.g., dangerous flight maneuvers or physical obstacles. There are two general approaches to evaluating these risks. One is a pessimistic approach or worst case control, where the worst possible outcome is optimized (Simon et al., 2011). This method, however, results in very constrained policies and might not be practical for most applications. Another approach is to define an acceptable risk level (Geibel & Wysotzki, 2005). An optimal policy is then defined as avoiding error states in general.

Multi-agent path planning for UAVs is one of the applications for risk-sensitive learning. (Geramifard, Redding, & How, 2013) propose such an approach, which combines adaptive modeling and reinforcement learning in the context of UAV mission planning to avoid the risks of running out of fuel during the mission. The risk is analyzed using a model, and it is defined as the probability of visiting any of the constrained states. Monte-Carlo simulation is used to estimate the associated risk, by simulating the trajectory from the current state. It is indicated that accuracy of the model has great influence on the optimality of the selected action, as an inaccurate model may lead to suboptimal policies. Another approach to path planning using RL methods is introduced in (Zhang, Mao, Liu, & Liu, 2013). This approach uses a novel method called Geometric Reinforcement Learning, similar to Q-learning, and it aims to solve the problem of limited information about possible risks by incorporating observed data from multiple UAVs, to avoid threat objects. Other factors of risk include safety distance between two UAVs. The optimal policy is defined as achieving the shortest possible path, combined with the lowest amount of risk.

While risk-sensitive learning had been successfully applied to tasks such as path planning, it is also applied to dynamical, nonlinear systems. Such an approach is described in (Geibel & Wysotzki, 2005). It presents another variation of the Q-learning algorithm and it is applied in a context of ensuring optimal operation of the distillation column. The risks are defined by tank level and substance concentrations, which are required to stay within specified limits, with a certain chance of constraining violation. High negative rewards are associated with the system entering an error state. The optimal policy then avoids the error states, but this approach also has some issues: it is unknown how large the risk of entering an error state is. The fact that it is impossible to avoid error states entirely is tolerated to some extent, but the risk is still constrained by some value selected beforehand. The approach does not use a model, but the agent is assumed to have access to simulated or empirical data. Learning is accomplished using empirical data obtained through interaction with a simulated or real process, and neural networks are used to apply the algorithm to processes with continuous state spaces. Preliminary experiments demonstrated in the paper shown that oscillations may occur between different sensible policies. These effects, however, could be prevented by using a discounted risk that leads to an underestimation of actual risk.

Overall, RL techniques for autonomous flight control show a lot of promise. There was a lot of progress in creating and refining RL methods for intelligent and adaptive control. However, to this day, RL techniques were only applied to solve very specific problems, like designing a controller valid for just one particular segment of the flight envelope or performing just one specific task (e.g., automatic landing, hover, etc.). And they mostly had been applied in theoretical/simulation models, with limited practical tests to validate these models.

Reinforcement Learning based framework development

The purpose of this research is to investigate the feasibility of reinforcement learning combined with safe exploration for the purpose of unmanned vehicle control and to develop a functional algorithm framework that could be applied to control of a UAV. The main reasons for designing such a controller are potential benefits with regard to unmanned vehicle safety, as well as increased autonomy. The performance of the resulting controller could be measured by it's stability, capability of obtaining an optimal solution, speed of learning and adaptability. This allows formulation of several research questions:

1. What is the current state of art research in the field of reinforcement learning and safe exploration for UAV?
   (a) What is the progress done in the field already?
   (b) What methods are commonly used?
   (c) What are the underlying issues that have to be dealt with when applying reinforcement learning to vehicle control?
   (d) What are the common principles of applying reinforcement learning to practical problems?
2. What are the candidate methods for intelligent policy improvement?

(a) Which methods had been applied already to control of UAVs?

(b) How developed are these methods, had they been applied in practice, or had they only been hypothesized in theory?

(c) What are the individual merits and limitations of the methods under consideration?

(d) What are the qualities of these methods e.g., real-time performance, computational requirements and convergence characteristics?

(e) Which methods are suitable for the purpose of this research?

3. What is the experimental set-up that could be used to evaluate the performance of the resulting controller algorithm?

(a) Is the resulting controller capable of its intended function: vehicle control?

(b) How does the resulting controller compare against existing conventional controllers?

(c) Is it practical to apply it to vehicle control?

To answer these questions, a Q-learning based controller is designed to steer the inner-loop dynamics of the UAV. These include the UAV pitch, roll, and yaw. The main methodology and the results are demonstrated in the paper, presented in Chapter 2. The paper describes an implementation of a reinforcement learning-based framework applied to the control of a multi-copter rotorcraft and discusses some of the results of algorithm evaluation using a simulated environment.

Chapter 3 outlines various reinforcement learning methodologies and techniques. It gives a brief introduction to the history of these methods and some of the core principles, given providing the background of the current state-of-art research. Some of the most popular methods such as the Q-learning and SARSA are discussed in detail. Chapter A outlines the proposed experiment. The experiment consists of a simulated quadrotor UAV model, controlled by an RL-based controller. Policy search is done offline, using a reduced order plant model, approximated using RBF neural net. Model inversion is used to convert policy actions into actuator inputs.

The states of the quadrotor are continuous, and the Q-learning methodology has to be adapted to deal with continuous states. It is possible to adjust it by applying function generalization to store the Q-function. Furthermore, various strategies can be used to extract a continuous range action from the resulting generalized policy. These techniques are introduced in chapter 4. Radial Basis Function (RBF) and CMAC networks are proposed as candidate generalization methods. Their advantages and disadvantages are discussed. Some particular phenomena such as the curse of dimensionality and effects of generalization on the resulting Q-function approximation are discussed. Some techniques designed to optimize the generalization are also introduced, including state pre-scaling and a hash-based RBF neural net approach.

The proposed control scheme deals with continuous state inputs, and it also produces continuous actions. There are various ways to extract a continuous value from the generated policy. Some of them are discussed in chapter 5. PSO method can deal with nonlinear function approximators such as CMAC. Another discussed method is based on sectioning of the

generalized Q-function, taking a projected "slice" of it and then approximating it using a combination of radial basis functions. Then the maximum is then approximated using Newton's root finding method.

The system dynamics are discussed in chapter 6. This includes an extended discussion on the mathematical background behind quadrotor modeling and a description of a dynamics model implementation in software. Additional specific discussed include the control allocation methodology, designed to avoid conflicts between system inputs. A full drone control scheme is proposed, including inner and outer loop controls. Virtual inputs are defined. And the reduced model identification is discussed. This reduced model is also used to create a model inversion-based controller, which is discussed in the chapter.

The resulting set of controllers is described and tested in chapter 7. The description is given for controller learning parameters and the reward function. Several tests are performed, using PID control scheme performance as a benchmark. The reinforcement learning controller is found to show performance comparable with that of a PID and to exceed it. Overall, the resulting control framework is demonstrated to apply to continuous state and action systems.

Chapter 2

Paper

Continuous state and action $\mathcal{Q}$-learning framework applied to quadrotor UAV control

Anton E. Naruta*

This paper describes an implementation of a reinforcement learning-based framework applied to the control of a multi-copter rotorcraft. The controller is based on continuous state and action $\mathcal{Q}$-learning. The policy is stored using a radial basis function neural network. Distance-based neuron activation is used to optimize the generalization algorithm for computational performance. The training proceeds off-line, using a reduced-order model of the controlled system. The model is identified and stored in the form of a neural network. The framework incorporates a dynamics inversion controller, based on the identified model. Simulated flight tests confirm the controller's ability to track the reference state signal and outperform a conventional proportional-derivative(PD) controller. The contributions of the developed framework are a computationally-efficient method to store a $\mathcal{Q}$-function generalization, continuous action selection based on local $\mathcal{Q}$-function approximation and a combination of model identification and offline learning for inner-loop control of a UAV system.

I. Introduction

In the recent decades, a lot of advancements had taken place in the field of reinforcement learning (RL).[1–3] In more general terms, reinforcement learning is a paradigm concerned with creating system controllers that aim to maximize some numerical performance measure, designed to achieve some long-term objective.[3] It is suited for nonlinear problems, optimal control, and it allows to combine qualitative and quantitative data.

Reinforcement learning techniques are often applied for identification of static systems, such as pattern recognition type problems.[4] RL methods are also often combined with more conventional control techniques, such as PID control, to optimize the controller parameters. More recently there had been an increased interest in applying reinforcement learning techniques for identification and control of dynamic systems, such as navigation of autonomous vehicles or robots.[5–8]

Control of unmanned vehicles, such as aerial drones, represents a challenging control problem. The dynamics of such systems are nonlinear, high dimensional and they contain hidden couplings. There are many types of unmanned aerial vehicles (UAVs), including lighter-than-air vehicles, fixed-wing aircraft, helicopters, and multi-copters. Multicopters have rather complex dynamics, and they are considered to be harder to control than fixed-wing aircraft.[6] They are open-loop unstable and require a skilled operator or a sophisticated autopilot to fly. Reinforcement learning can be applied to design an optimal, adaptive controller for such vehicles.

Here we introduce a reinforcement learning framework for automatically synthesizing a controller for any generic multi-copter UAV, using reinforcement learning methodology. The proposed framework consists of a combination of model identification of the agent dynamics and off-line $\mathcal{Q}$-learning. The identified model is used to train the policy, and it is also used to create a dynamics inversion controller to translate the policy actions into direct inputs to the vehicle. The policy is stored in the form of Radial Basis Function (RBF) neural net, optimized for real-time performance. The resulting policy works with continuous agent states and produces continuous actions. The controller is adaptive by design, and it is capable of dealing with changes in agent behavior and adjusting itself online.

The performance of the controller is compared to the performance of a conventional, PID-based controller. It is shown that the performance of the RL controller is comparable to that of conventional control schemes, and exceeds it. Additionally, it is demonstrated that the RL controller is capable of adjusting its behavior online if agent behavior deviates from the initial model used to train the policy.

*Control & Simulation dept., Aerospace Engineering, TU Delft

II. Reinforcement learning

Reinforcement learning and intelligent control are used increasingly in various control applications. Reinforcement learning can deal with the nonlinearities that occur in most real-life systems, and it is highly adaptable and adaptive. As such, it had been studied and applied to a wide range of problems in a variety of different fields. In reinforcement learning the system consists of an *agent*, acting in some *environment*. For example, an agent could be an autonomous vehicle, while the environment is the world in which it acts. The learning process itself consists of the agent taking different actions to *explore* the system and to *exploit* it by using the knowledge that it already has.

After each transition from agent state s to s' the agent receives some reward r. If successful, the agent receives a higher reward, and if the action leads to an undesirable state, then the magnitude of the reward is lower. This reward takes the shape of a certain *reward function*, determined beforehand. In autonomous vehicle context, the rewards could be defined by smooth movements, quick response, the safety of the operation, etc. The purpose of an agent is to maximize this reward by taking optimal actions. By acting in a certain way, the agent follows a *policy* π, which represents the mapping from perceived states of the environment to actions to be taken.[3] While the reward function $r(s)$ determines the actions of an agent in a short run, influencing the policy taken, a *value function* $V(s)$ is used to specify it's total reward, in the long run, determining the agent's behavior B. This behavior can be learned using a wide variety of different algorithms. Formally, the reinforcement model consists of:

- a set of environment states, $\mathcal{S}$, which can be discrete or continuous
- a set of actions $\mathcal{A}$, which can be discrete or continuous
- a set of reinforcement signals r, which can be static or varying, depending on agent state

A. Markov decision process

In addition to immediate rewards, a reinforcement learning agent must be able to take future rewards into account. Therefore it must be able to learn from delayed reinforcement. Typically the agent would progress throughout it's learning process, starting off with small rewards, and then receiving larger rewards as it gets closer to its goal. And it must learn whichever actions are appropriate at any instance, based on rewards that the agent will receive in the future. Delayed reinforcement learning problems can be modeled as *Markov decision processes*(MDP).[3] An MDP consists of

- a set of states $\mathcal{S}$
- a set of actions $\mathcal{A}$
- a reward function $R : \mathcal{S} \times \mathcal{A} \to \Re$
- a state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$, where Π maps the states to probabilities. The probability of transition from state s to state s' given action a.

The state transition function specifies the transition from one state to another, following an action was taken by the agent. The agent receives an instant reward after performing an action. The model has *Markov* property if the state transitions are independent of any previous environment states or agent actions.

B. Reward function

The rewards could be either positive or negative, which has an impact on agent behavior, at least during initial learning stages. Assuming the $\mathcal{Q}$-function is initialized at 0, updating it with a positive reinforcement after each episode iteration results in a high value assigned to the action taken, compared to all other possible actions that are assigned a zero value at initialization. As a result, the agent will tend to select actions that it had applied previously for as long as the outcome of other actions is still uncertain. This heuristic is called an optimistic approach. As a result of applying it, a strong negative evidence is needed to eliminate an action from consideration.[1] In negative approach only negative reinforcement takes place. As a consequence the agent tends to explore more, eliminating actions that result in lower rewards and eventually settling on the

best policy. The negative reinforcement approach was selected. The chosen reward function is a combination of the state offset from some desired state reference, current state value and state derivative value:

$$r = \dot{s} - sign(\Delta s) * \sqrt{sign(\Delta s) * \Delta s}, \tag{1}$$

where Δs represents the difference between controlled state s and the reference state $\Delta s = s - s_{ref}$.

C. $\mathcal{Q}$-learning

One of most important reinforcement learning algorithms available today is $\mathcal{Q}$-learning.[3] It is an off-policy TD control algorithm. Off-policy means that it learns the action-values that are not necessarily on the policy that it is following. The $\mathcal{Q}$-learning is using a $\mathcal{Q}$-function to learn the optimal policy. The expected discounted reinforcement of taking action a at state s is represented as $\mathcal{Q}^*(s, a)$.

$$\mathcal{Q}^*(s) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') \max_{a'} \mathcal{Q}^*(s', a') \tag{2}$$

This function is related to value as $V^*(s) = \max_a \mathcal{Q}^*(s, a)$. Then the optimal policy is described as $\pi^*(s) \arg\max_a \mathcal{Q}^*(s, a)$. One-step $\mathcal{Q}$-learning is defined by the following value update rule:

$$\mathcal{Q}(s, a) \leftarrow \mathcal{Q}(s, a) + \alpha \left[r + \gamma \max_{a'} \mathcal{Q}(s', a') - \mathcal{Q}(s, a) \right] \tag{3}$$

In this equation, $\mathcal{Q}$ stands for action-value function, updated at every step at a learning rate α. The update parameters are the reward r and the value of $\mathcal{Q}$ at the next step, corresponding to the maximum possible cumulative reward for a certain action a, given that policy π is followed. Factor γ is the discount rate; it represents the horizon over which the delayed reward is summed, and it makes the rewards earned immediately more valuable than the ones received later.

There are numerous variations to $\mathcal{Q}$-learning algorithm, aiming at improving convergence and optimality characteristics, and trading off exploration and exploitation during the learning process.

SARSA algorithm is an on-policy variation of $\mathcal{Q}$-learning.[1] An action-value function has to be learned rather than a state-value function. For an on-policy method $\mathcal{Q}^\pi(s, a)$ must be estimated. This can be done using a nearly identical kind of update as in the general $\mathcal{Q}$-learning. SARSA learning update rule is defined as follows:

$$\mathcal{Q}(s, a) \leftarrow \mathcal{Q}(s, a) + \alpha \left[r + \gamma \mathcal{Q}(s', a') - \mathcal{Q}(s, a) \right] \tag{4}$$

The difference with the $\mathcal{Q}$-learning update, described earlier is that rather than the optimal $\mathcal{Q}$-value $\max_a \mathcal{Q}(s', a)$ the value of the action selected for the consecutive step $\mathcal{Q}(s', a')$ is used during the update.

III. Continuous state and action $\mathcal{Q}$-learning

$\mathcal{Q}$-learning algorithms are commonly applied to problems with discrete sets of states and action. In those, the state-action spaces and rewards are encoded in tabular form, where each cell represents a combination of state and action. In reality, however, control problems where the states and actions are continuous are also very common.[5] Adapting the "classical" $\mathcal{Q}$-learning approach to deal with these types of problems presents a challenge. A common approach is to discretize the state of the world by splitting the state domain into discrete regions. This method, however, introduces some problems. If the state is coarsely discretized a *perceptual aliasing* problem occurs.[9] It is difficult to discretize the world states without losing information. One of the solutions is to discretise the world more finely. This increase of the resolution, also increases the amount of memory required, introducing the curse of dimensionality.[10] Fine discretization may lead to an enormous state-action space, which in turn may result in excessive memory requirements and difficulty in exploring the entire state-space.

These issues could be solved by using *generalization* e.g., using experience with a limited subset of the state-space to produce an approximation over a larger subset.[3] Generalization allows applying the experience gained from previously visited states to the ones that hadn't been visited yet.

One type of generalization is *function approximation*, which in turn is an example of supervised learning. Most supervised learning methods seek to minimize the mean-squared-error (MSE) over a distribution of inputs $P(s)$.

There are various methodologies available to approximate a continuous $\mathcal{Q}$-function. *Neural Networks* (NN) can be used to model and control very complex systems with sufficient accuracy without complete knowledge of its inner composition. Neural networks themselves can be divided into two types: multilayer neural networks and recurrent networks. Despite some differences, they could be viewed in a unified fashion as part of a broader discipline .[4] Another way to approximate a continuous function is using *coarse coding*. This is a technique where the state-space region is separated into overlapping regions, each representing a feature. In the binary case, if the state is situated inside of a feature then the corresponding feature has a value of 1 and is said to be *present*, otherwise, this feature is 0 and it is said to be *absent*.[3] The parameters that affect the discretization accuracy of this method are the size of the regions and number of overlapping regions.

A. Neural Networks

Warren McCulloch and Walter Pitts proposed the concept of neural networks first. They came up with a model of an artificial neuron, later dubbed *perceptron*.[11,12] The basic building unit of a neural network is an artificial neuron. It consists of a weighted summer, the function of which is to combine one or several inputs. Each input is multiplied by some weight, which can be pre-determined, or adjusted as the learning process goes on. The combined and weighted inputs then pass through some nonlinear activation function. Various types of activation functions exist. A simple threshold function produces a unit output when some input threshold is passed. A bell-curve shaped radial-basis function gradually increases or decreases, depending on the input value proximity to the center of the neuron. A tangent-sigmoidal activation function acts similar to a threshold function but produces smooth, varying output.

Multiple neurons can be combined in layers, and layers themselves could be combined as well. A typical neural network would have one hidden layer, that consists of several non-linear neurons, and one output layer that often consists of a simple summator of weighted hidden layer outputs. The layout of a basic neural net is shown in Figure 1.

One of the popular choices for neuron activation function is the Gaussian Radial Basis Function (RBF). Each neuron in the hidden layer of an RBF net has the following activation function:

$$\phi_j(\nu_j) = a_k \cdot \exp(\nu_j) \tag{5}$$

The coefficients a_k are the output weights of the network, and it controls the amplitude of the RBF. The input of the activation function is the distance between the location of the input data point and the center of each neuron:

$$\nu_j = -\sum_i w_{ij}^2 (x_i - \mathbf{c}_{ij})^2 \tag{6}$$

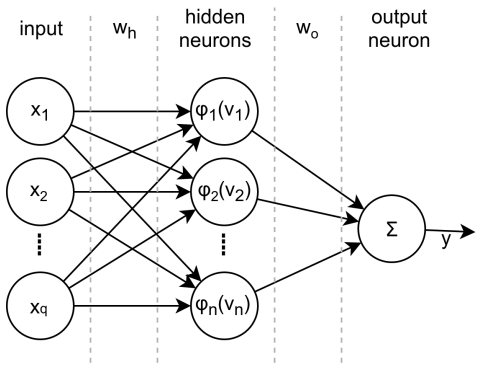


Figure 1. Neural net layout

The value $w_{i,j}$ corresponds to the width of the RBF, while $\mathbf{c}_{i,j}$ corresponds to the center of the activation function at each neuron. There are various ways to initialize all the outlined parameters. The neurons themselves could be either placed randomly, or they could also be organized in a grid. There are a variety of training methods that apply to NN, e.g. linear regression methods, and gradient descent methods such as back-propagation,[4] Levenberg-Marquardt,[13] and others.

NN can be relatively expensive to implement computationally. The overhead can be reduced by only considering the neurons activated at a time and approximating the exponential function using a polynomial or a lookup table with pre-calculated values.

B. State pre-scaling

The network inputs can be pre-scaled before performing a forward pass. This allows specifying a focus region for the network with higher resolution, which can be beneficial to the application because it allows saving on the number of weights. Equation (7) shows a case of using a root function to encode an input variable. There, the input variable $x \in [a, b]$ is scaled as $x_s \in [0, 1]$

$$x_s = \begin{cases} \frac{1}{2}\left(1 - \sqrt[r]{\frac{h-x}{h-a}}\right) & \text{if } x < h \\ \frac{1}{2}\left(1 + \sqrt[r]{\frac{x-h}{b-h}}\right) & \text{if } x >= h \end{cases} \tag{7}$$

The variable x_s denotes a scaled version of a variable x. The offset factor h denotes the "focus" of the scaling, e.g. the values for which the scaling results in a grid that is finer. This sets the region around which the scaling gradient is the steepest. The variable r denotes the scaling power. When it's set to one, the scaling is purely linear.

C. Encoded RBF-based neural net

Tile coding is a form of coarse coding in which the receptive fields of features are grouped into an exhaustive partition of the input space. These groups are referred to as tilings, and each element is called a tile. This idea is an integral part of Cerebral Model Articulation Controller (CMAC) method .[14] The feature values are stored in a weight vector table where each cell has it's association cell that indicates binary membership degree of each possible state input vectors. The activated weights of a fully trained CMAC mapping are summed across to calculate the resulting output. Since the only algebraic operation that takes place is the addition, CMAC is a very computationally efficient method. The number of tilings determines the computational effort: denser distributions require more calculations and are more computationally intensive.

The CMAC has an advantage over a "pure" implementation of radial basis function neural network, where all weights are activated with each forward pass and all weights are updated at each backward pass. In the CMAC only the local weights, situated close to the state, are activated and updated, and finding it's output is as simple as summing the weights without any need for using an activation function. Nevertheless, the nature of the activation function used in an RBF net makes it possible to optimize it for computational performance using a hashing methodology similar to CMAC.

Equation 5 describes a typical Gaussian Radial Basis Function (RBF). RBF is based on the principle that its output value gradually drops as the input state moves further away from the neuron center. This property also means that at a certain distance from the neuron center the neuron output contribution is negligible, compared to the neurons activated closer to the state value. Therefore, the neurons situated further away from the activated region can be neglected, since they have little influence on the output of the neural net in that area. By limiting the number of activated neurons the size and resolution of the network can be decoupled from real-time performance. If only a limited number of neurons is activated, it does not matter anymore how many neurons there are in total. If the neurons are evenly spaced throughout the state range, selecting the neurons that have to be activated becomes relatively straightforward.

In order to encode continuous state values into neuron coordinates first define index i. Discrete index i corresponds to tile coordinate of the neuron closest to the input. Given a variable x, the floor operator $\lfloor x \rfloor$ is defined as:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\} \tag{8}$$

In equation (8) the value of $\lfloor x \rfloor$ can take negative or positive values. But the neuron index i has to be compatible with positive-only indexing systems $\{i \in \mathbb{Z} \mid 0 \leq i \leq M\}$. Furthermore, given $M-1$ neurons per variable dimension on the interval $[a, b] = \{x \mid a \leq x \leq b\}$ the variable has to be scaled This scaling can be done in a variety of ways, depending on the application. The neuron index i can be calculated given a continuous variable value x on the interval $[a, b]$, using the scaling method described by 7:

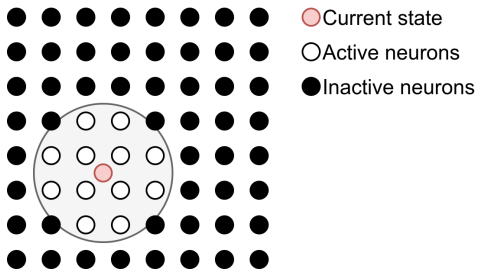


Figure 2. Neuron activation scheme

$$i \quad = \quad \lfloor M \times x_s + 0.5 \rfloor \tag{9}$$

The index variable i stands for the index of the neuron nearest to the scaled input x_s. As Figure 2 illustrates, it is possible to activate only the neurons situated within some maximum radius $r_{lattice}$. The distribution of the activated neurons always follows the same type of circular lattice pattern, offset from the

coordinate of the current input state. This pattern can be calculated previously to network initialization, stored and re-used. These patterns are stored as an offset from the neuron, situated the closest to the input state, denoted as $\bar{c}_{i,j,\ldots,n}$, where n denotes the network input dimensionality.

The following encoding principle can be illustrated for a single-input coded neural network with activation radius $r_{lattice}$ with N hidden neurons uniformly spread through the range $[a, b]$, where $a \leq x \leq b$. The lattice offset coordinates for such a network can be represented as

$$c_{lattice} = [-R, \ldots, 0, \ldots R] \tag{10}$$

$$R = \lfloor r_{lattice} \rfloor \tag{11}$$

Given an input x, the coordinate of the nearest neuron i can be found using 9. Then the activated neuron coordinates and the neuron input offsets $\Delta_i = x - \mathbf{c_i}$ become:

$$c_{neur} = i + [-R, \ldots, 0, \ldots, R] \tag{12}$$

$$\Delta = (x_s - i) + [-R, \ldots, 0, \ldots, R] \tag{13}$$

The entire procedure is quite simple and allows to calculate the neuron coordinates as well as inputs for each neuron simultaneously and efficiently. This methodology could also be easily extended to higher input dimensions by generating multi-dimensional neuron offset lattices. With each additional dimension, the number of required neurons grows. The indexing methodology can be extended to facilitate encoding of multi-dimensional values of $\bar{x} = [x_1, x_2, \ldots, x_{n-1}, x_n]$, with n-dimensions.

Table 1 shows the number of activated neurons for a value of $r_{lattice} = 2.4$ and the percentage of the total neurons in the network for an encoding that consists of 8 neurons per-dimension.

		N_a activated neurons (% total)			
N_{inputs}	$N_{neurons}$	$r = 1.0$	$r = 1.5$	$r = 2.0$	$r = 2.4$
1	8	3 (37.5 %)	3 (37.5 %)	5 (62.5 %)	5 (62.5 %)
2	64	5 (7.81 %)	9 (14.0 %)	13 (20.3 %)	21 (32.8 %)
3	512	7 (1.37 %)	19 (3.71 %)	33 (6.44 %)	57 (11.13 %)
4	4096	9 (0.22 %)	33 (0.81 %)	89 (2.17 %)	137 (3.34 %)

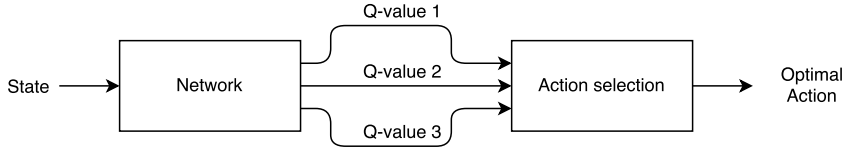
Table 1. Number of activated neurons for selected dimensions

The number of activated neurons grows with increased radius and input dimensionality. For a single-input neural net, the effect of neuron encoding is quite small, and the extra computational load might offset any benefits. For a larger number of inputs, however, there is a drastic increase in efficiency. For a neural net with three inputs, encoded with eight neurons per-dimension and activation radius $r = 2.4$ only 11.13 % of all neurons have to be activated at any forward or backward pass. Increasing the resolution does not affect the computational load since only a fixed number of neurons is activated and the effect is more pronounced for a higher number of inputs.

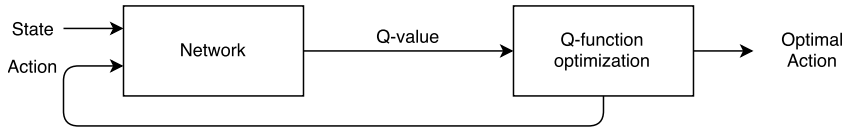
D. Optimal action selection

The underlying principle of $\mathcal{Q}$-learning requires a method that is capable of finding an optimal action, optimizing the output value of the $\mathcal{Q}$-function. Various methodologies can be applied, depending on the type of generalization used to store the $\mathcal{Q}$-function approximation. Given a continuous-state $\mathcal{Q}$-learning problem, there are two basic approaches to find the optimal action: using a discrete set or a continuous range of actions. A schematic in Figure 3 illustrates these two methods.

In a discrete-action scenario the network used to generalize the $\mathcal{Q}$-function produces several outputs for any current state, each of them associated with a predetermined action value. Once these values become available, the action that results in the highest output is selected as the optimal one. In a continuous approach, the network inputs include both the state and the action. The output value is then optimized to find the optimal action using some optimization algorithm. The discrete approach has an advantage over a continuous one because when only one forward pass of the network is required, whereas the continuous method typically requires several passes. A limited number of forward passes means it's less demanding



(a) Discrete action



(b) Continuous action

Figure 3. Q-learning action selection approaches

computationally and easier to implement. It's still possible to produce a continuous action when using a discrete approach, using some interpolation technique.[5] On the other hand, a continuous method allows to select the action more precisely, and the actions themselves transition more smoothly. Smooth, continuous inputs offer an advantage when controlling a sensitive system such as a quadrotor.

A continuous-action approach was selected for the designed RL framework. This section outlines some optimization techniques applicable to action selection. Consider a function $y(x)$ in the form

$$y(x) = g(x)\theta \tag{14}$$

where $g(x)$ represents a set of hidden layer neuron outputs, as described in A, for a given input value x

$$g(x) = \begin{bmatrix} \phi_0(x) & \phi_1(x) & \dots & \phi_N(x) \end{bmatrix}, \tag{15}$$

$$\phi_i(x) = \exp(-(x - \mathbf{c}_i)^2), \tag{16}$$

and θ is the parameter vector that contains network output weights $\theta = [w_0, w_1, \dots, w_N]$. This function can be used to describe a local action-value distribution for a given state. The activation function described by 16 can be differentiated twice with respect to input x:

$$\dot{\phi}_i(x) = -2(x - \mathbf{c}_i)\phi_i(x) \tag{17}$$

$$\ddot{\phi}_i(x) = -2(1 - 2(x - \mathbf{c}_i)^2)\phi_i(x) \tag{18}$$

This allows to express the derivatives of functions 17 and 18:

$$\dot{g}(x) = -2 \begin{bmatrix} (x - \mathbf{c}_0) & (x - \mathbf{c}_1) & \dots & (x - \mathbf{c}_N) \end{bmatrix} \times g(x) \tag{19}$$

$$\ddot{g}(x) = -2 \begin{bmatrix} 1 - 2(x - \mathbf{c}_0)^2 & 1 - 2(x - \mathbf{c}_1)^2 & \dots & 1 - 2(x - \mathbf{c}_N)^2 \end{bmatrix} \times g(x) \tag{20}$$

Note that the exponential component in 16 and the offsets $(x - \mathbf{c})$ can be re-used when finding the derivatives of the activation function, which means that the computational load can be reduced when implementing these calculations in software.

For the function in 14 to reach it's minumum/maximum the output function derivative must reach zero:

$$\dot{g}(x)\theta = 0, \tag{21}$$

This condition does not have an analytical solution. Here, Newton's iteration method can be applied. Newton's iteration is a numerical method designed to find roots(zeros) of differentiable equations in the form

$$f(x) = 0 \tag{22}$$

Provided an initial guess x_0, the successive update formula for this method is

$$x_{k+1} = x_k - \frac{\dot{f}(x_k)}{f(x_k)}, k = 0, 1, \ldots \tag{23}$$

Typically with each consecutive update, the output of the function $f(x_k)$ converges closer to its nearest zero value. This mechanism, however, is only capable of finding the nearest root, one at a time. Therefore, for a function with several possible zeros, several guess values must be supplied.

When applied to the problem of finding the $\arg\max_a \mathcal{Q}(s, a)$ the Newton's iteration procedure can be applied as follows: First, the $\mathcal{Q}$-function is interpolated at state s. The original $\mathcal{Q}$-function approximator would typically have several state inputs and one action input. At a given state all inputs other than the action are fixed and only the action input can be varied. Therefore it is possible to take a 1-dimensional "slice" of the $\mathcal{Q}$-function, by sampling the $\mathcal{Q}$-function output with different actions $\bar{a} = \Big[a_0, a_1, \ldots, a_M\Big]$. Outputs of the $\mathcal{Q}$-function are recorded in $\bar{Y} = [\mathcal{Q}(s, a_0), \mathcal{Q}(s, a_1), \ldots, \mathcal{Q}(s, a_M)]$. The parameter vector $\bar{\theta}$ can be calculated using linear regression:

$$\bar{\theta} = (A(\bar{x})^T A(\bar{x}))^{-1} A^T \bar{Y}, \tag{24}$$

where each row i of A contains the inner-layer outputs of the interpolant neural net $\Big[\phi_0(a_i), \phi_1(a_i), \ldots, \phi_n(a_i)\Big]$ for the supplied action value a_i. The first and the second derivatives of the approximation function $\bar{y}(a) = g(a)\bar{\theta}$ are:

$$\dot{\bar{y}}(a) = \dot{g}(a) \cdot \bar{\theta} \tag{25}$$

$$\ddot{\bar{y}}(a) = \ddot{g}(a) \cdot \bar{\theta} \tag{26}$$

Newton's update rule to find the zeros of $\dot{y}(a)$:

$$a_{k+1} = a_k - \frac{\ddot{y}(a_k)}{\dot{y}(a_k)}, k = 0, 1, \ldots \tag{27}$$

After evaluating several starting points the action value a several potential minima/maxima of the optimized function. Then the action value that results in the maximum/minimum output of the function $\mathcal{Q}(s, a)$ can be found by comparing the outputs of $\bar{y}(a)$ and finding the maximum value.

IV. UAV dynamics and control scheme

The Quadrotor concept had been known since the early days of aeronautics. One of the earliest examples of such aircraft is the Breguet-Richet Quadrotor helicopter Gyroplane No.1, built in 1907. A quadrotor is a vehicle that has four propeller rotors, arranged in pairs. The two pairs (1,3) and (2,4) turn in opposing directions and variation of the thrust of each rotor is used to steer the vehicle pitch, roll, yaw, and altitude. This section introduces some modeling methods that can be used to simulate a quadrotor vehicle and presents a general mathematical model of quadrotor dynamics.

A. Quad-copter modelling and control

The control scheme of a conventional aerial quadrotor vehicle consists four principal parts: pitch and roll controls, collective thrust and yaw control. Quadrotors have six degrees of freedom: three translational and three rotational, and they can be controlled along each of them. Rotational and translational motions are coupled: tilting along any axis results in vehicle movement in the direction perpendicular to the tilt angle. Extra couplings exist between individual rotors and the aircraft body. For example to rotate along the vertical axis while maintaining altitude the distribution of thrust has to change in such manner that some of the rotors spinning in one direction spin slower, while the rotors that are turning in the direction of intended rotation are moving faster. These various couplings result in a challenging problem: in practice, it 's hard to control a quadrotor purely manually. In addition to the difficulty of control, there is also a question of stability of multi-copter craft. Multi-copters are not inherently stable; it is necessary to apply active damping. Electronics are used to stabilize it and distribute the thrust across the rotors.

B. Dynamic model of a quadrotor

This section will describe modelling of a quad-copter using a purely white-box approach. Schematic of a quadrotor body and inertial frames is shown in Figure 4.

The craft dynamics can be modeled using Lagrangian method.[15,16] The vector containing generalized coordinates of the quadrotor can be defined as $q = (x, y, z, \phi, \theta, \psi) \in R^6$. It can be split into two components, for translational and rotational motion. The translational motion of the craft can be described using equations 28-30. The rotational motion is described using equations 31-33. System inputs are defined by equations 34 - 38. Multicopter actuators can be modelled as DC motors.[15] The motor dynamics are described in equation 39.

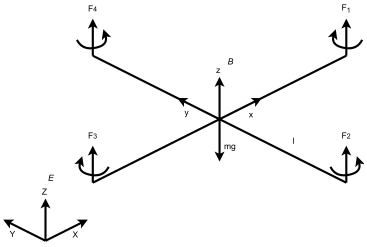


Figure 4. Quadrotorotor configuration schematic, body fixed frame B and inertial frame E.

$$\ddot{x} = \frac{s(\phi)s(\theta)s(\psi) + s(\phi)s(\psi}{m})U_1 \tag{28}$$

$$\ddot{y} = \frac{s(\phi)s(\theta)s(\psi) + s(\phi)s(\psi}{m})U_1 \tag{29}$$

$$\ddot{z} = -g + \frac{s(\phi)s(\theta}{m})U_1 \tag{30}$$

$$\ddot{\phi} = \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right) - \frac{J_p}{I_x}\dot{\theta}\Omega + \frac{l}{I_x}U_2 \tag{31}$$

$$\ddot{\theta} = \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right) + \frac{J_p}{I_y}\dot{\phi}\Omega + \frac{l}{I_y}U_3 \tag{32}$$

$$\ddot{\psi} = \dot{\phi}\dot{\theta}\left(\frac{I_x - I_y}{I_z}\right) + \frac{1}{I_z}U_4 \tag{33}$$

$$U_1 = C_T(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \tag{34}$$

$$U_2 = C_T(\Omega_4^2 - \Omega_2^2) \tag{35}$$

$$U_3 = C_T(\Omega_3^2 - \Omega_1^2) \tag{36}$$

$$U_4 = C_T(\Omega_1^2 + \Omega_3^2 - \Omega_2^2 - \Omega_4^2) \tag{37}$$

$$\Omega = \Omega_2 + \Omega_4 - \Omega_1 - \Omega_3 \tag{38}$$

$$\dot{\omega} = \frac{K_e}{RJ_t}u - \frac{K_e^2}{RJ_t}\omega - \frac{d}{J_t}\omega^2 \tag{39}$$

where:

State	Unit	Description	Constant	Value	Unit	Description
x	[m]	x-coordinate	m	1.2	[m]	Drone mass
y	[m]	y-coordinate	I_x	8.5e-3	[kg·m^2]	Moment of inertia x-axis
z	[m]	z-coordinate	I_y	8.5e-3	[kg·m^2]	Moment of inertia y-axis
ϕ	[rad]	Roll angle	I_z	15.8e-3	[kg·m^2]	Moment of inertia z-axis
θ	[rad]	Pitch angle	C_T	2.4e-5	[-]	Thrust factor
ψ	[rad]	Yaw angle	C_D	1.1e-7	[-]	Drag constant
Ω	[rad/s]	Rotor speed	R	2.0	[Ω]	Motor resistance
ω	[rad/s]	Motor angular speed	J_r	1.5e-5	[kg·m^2]	Rotor inertia
u	[V]	Supplied motor voltage	J_m	0.5e-5	[kg·m^2]	Motor inertia
			l	2.4e-1	[m]	Motor arm length
			K_e	1.5e-2	[-]	Motor constant

Table 2. UAV system states and relevant constants

Table 2 shows the set of parameters used for the simulation. These values were selected to achieve a realistic behavior of the model.

The full drone controller consists two parts: the inner loop, the and outer loop control. Lower-level inner loop controller is designed to track reference pitch, roll, yaw and altitude signals. The actuator delay due to motor dynamics influences the UAV response behavior. Higher-level outer loop controller is designed to track a position reference, such as x and y coordinates of the vehicle, and perform maneuvers. The external

loop controllers pass on reference signals to the inner loop, and they are set up in a cascaded fashion. Figure 5 shows an example of a basic drone control scheme layout.

There are four inner-loop controllers in place: roll, pitch, yaw and altitude control. There are also two outer loop position controllers, for states x and y. The operator interacts with the controller via an interface. Reference signals for pitch and roll φ_{ref}, θ_{ref} can be either set directly through the interface, or as a command from higher-level x and y controllers. Yaw angle and altitude reference signals ψ_{ref} and z_{ref} are set directly. It is also possible to add an additional layer of controllers, that combine flight heading and yaw control or performs some set of maneuvers, for example. The four inner-loop controllers produce scaled control inputs v_b (altitude), v_{02} (pitch controller), v_{13} (roll controller), and v_{0213}. Inner-loop controller inputs and outputs are defined as:

Controller	Inputs		Virtual control states
Altitude	z_{ref}, z, $\dot{z}$, $c(\phi)c(\theta)$	v_b	$= v_0^2 + v_1^2 + v_2^2 + v_3^2$
Roll	ϕ_{ref}, ϕ, $\dot{\phi}$, $\dot{\theta} \times \dot{\psi}$	v_{13}	$= v_1^2 - v_3^2$
Pitch	θ_{ref}, θ, $\dot{\theta}$, $\dot{\phi} \times \dot{\psi}$	v_{02}	$= v_0^2 - v_2^2$
Yaw	ϕ_{ref}, ϕ, $\dot{\phi}$, $\theta \times \psi$	v_{0213}	$= v_0^2 + v_2^2 - v_1^2 - v_3^2$

Note that the virtual control inputs still have to be processed to produce usable motor voltage inputs $v_0 \dots v_3$. Each voltage is limited, and different controllers might be in conflict with one another. This conflict between actuators makes it necessary to include a control allocation module, to interpret the resulting voltage differentials as motor inputs.

For a typical UAV, several controllers must be implemented, governing the pitch, roll, yaw and vertical motion dynamics. Figure 5 illustrates a schematic of how these controllers are combined to achieve full control of the UAV.

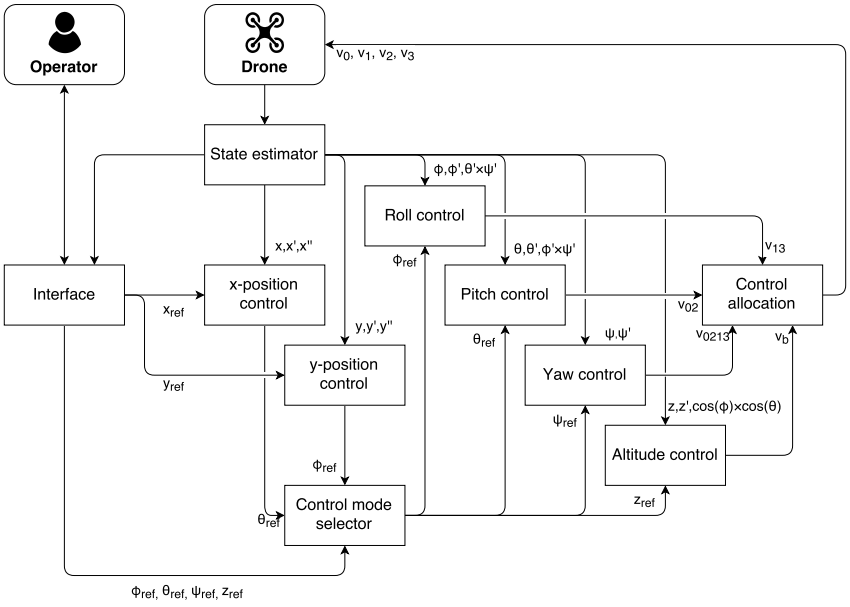


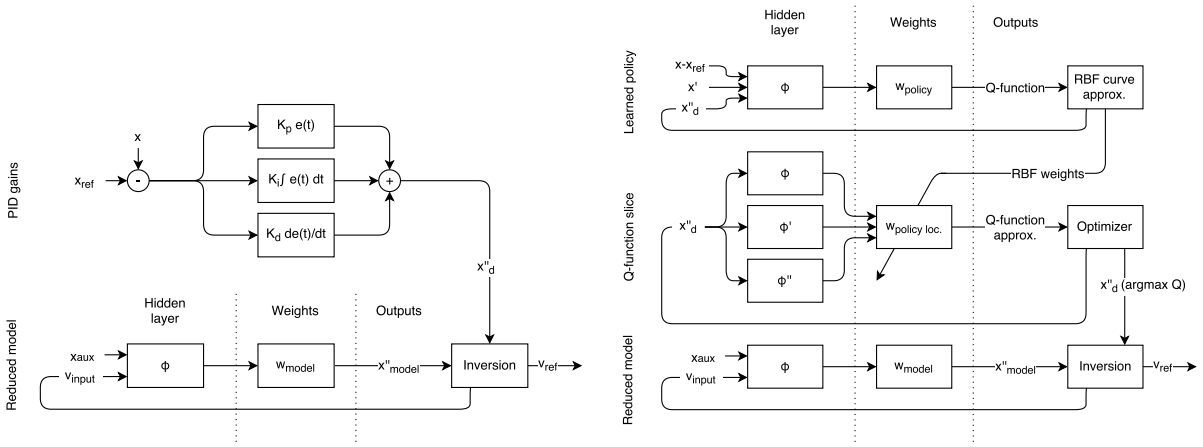
Figure 5. Drone control scheme

V. Reinforcement-learning based control of the UAV

Two controllers are designed to control the UAV: a conventional PD and a reinforcement learning-based approach. The PD controller performance serves as a benchmark for the RL-based solution, and it is used to validate the results. The PD gains are optimized using the PSO algorithm.[17–19] The RL-based controller is based on $\mathcal{Q}$-learning, described in II. Both controllers incorporate an adaptive model-based inversion dynamics inversion scheme to generate actuator inputs, as described in IV. This section describes both the conventional and the reinforcement learning based approaches.

A. Controller layout

The layout of the control scheme consists of two parts: the inner and the outer control loops. Two types of controllers are developed to fill the role of inner-loop control of the UAV: a conventional PD and a reinforcement learning approaches.



(a) PID control scheme

(b) RL control scheme

Figure 6. Conventional and RL-based control aproaches

Figure 6A illustrates a conventional feedback PID controller. It processes the signal reference offset $e(t) = x - x_{ref}$. The absolute value of this offset is multiplied with the proportional gain K_p, the integral of this error is multiplied with K_i and the rate of change of this error is multiplied with gain K_p. The output of the PID controller is the desired acceleration $\ddot{x}_d$. The model inversion module is then tasked with generating a suitable virtual input to achieve this desired acceleration.

PID controllers are relatively easy to implement and are sufficient for most real-world control applications. It can serve as a validation for reinforcement learning-based controller behavior.

The proposed $\mathcal{Q}$-learning controller is a lot more complex than the PID one. The reinforcement learning controller consists of policy, stored in the form of encoded RBF neural net, as described in A. Like with the PD controller, x, $\dot{x}$ and $\ddot{x}_{ref}$ are used as inputs. The difference comes from how the desired control inputs are produced: in a reinforcement learning controller the actions are generated by maximizing the output value of the network.

The policy is stored in the form of an RBF neural net that accepts at least three inputs: the offset between the current and the reference states $\Delta x = x - x_{ref}$, current state derivative $\dot{x}$ and the desired state acceleration $\ddot{x}_d$. The variable $\ddot{x}$ defines the action that can be taken by the agent. When states Δx and $\dot{x}$ are sampled at some point in time the action variable $\ddot{x}_d$ becomes the only unconstrained input of the generalized $\mathcal{Q}$-function. Therefore it becomes possible to make an instantaneous "slice" of the $\mathcal{Q}$-function, which can be described as a one-dimensional RBF-based curve. In order to do this, the output of the neural net that describes the $\mathcal{Q}$-function is sampled at variable action values $[x_{d,0}, x_{d,1}, \ldots, x_{d,N}]$. The outputs $Y = [Q_0, Q_1, \ldots, Q_N]$ are stored and used to generate a one-dimensional RBF-based function, using the procedure, described in Section **??**.

B. Reduced model and dynamics inversion control

The complete UAV dynamics, outlined int the previous section, consists of six states describing drone position and attitude(x, y, z, ϕ, θ, ψ), four rotor speeds (ω_0, ω_1, ω_2, ω_3) and four virtual inputs (v_b, v_{02}, v_{13}, v_{0213}) that can be translated into direct motor voltage inputs (v_0, v_1, v_2, v_3). All motions that the UAV goes through, along or about the x-axis, y-axis or z-axis are cross-coupled with other motions of the aircraft. Pitch, roll and yaw dynamics are all inter-connected, while the translational motions are connected to the current aircraft tilt offset from the vertical z-axis. This relatively complex model can be reduced into a more simple one. It could serve two functions: to enable inversion-based control and to be used for off-line reinforcement learning.

Some of the dynamics only play a marginal role in the overall performance of the controller. These dynamics include the response delay of the actuators(rotors) and the cross-coupling modes. The delay between the change of voltage supplied to the rotors, and the reaction of the system is very brief. This delay can be disregarded for a large enough time step. The cross-coupling effects, in turn, should largely be overshadowed by the contribution of control actuators during moderate flight maneuvers. More aggressive maneuvering might result in a larger deviation.

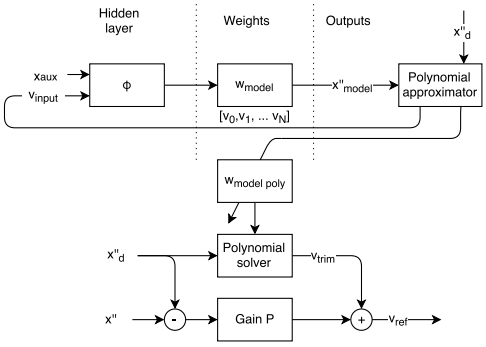
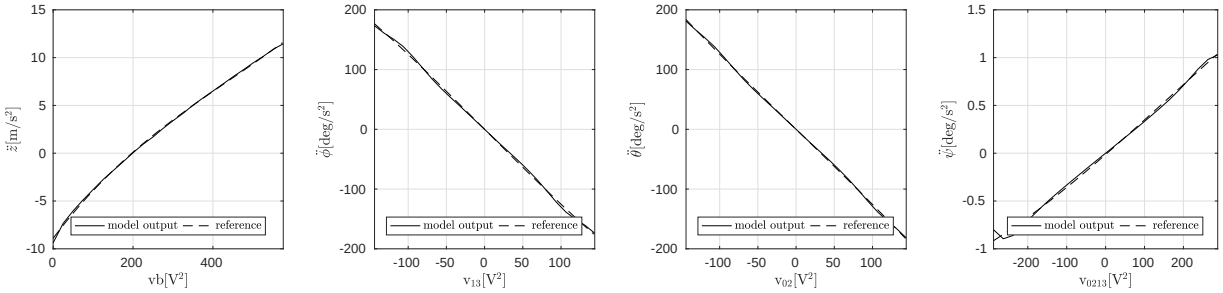


Figure 7. Inversion controller

Any cross-couplings, present in the pitch roll and yaw modes, are assumed to have no influence, and applying a virtual input to the system results in an immediate change of state, without any delay. The full dynamics model is split into four simpler sub-models, describing pitch, roll, yaw and altitude motion. An RBF-net is trained to model these dynamics. Each RBF net maps the relationship between virtual inputs and output acceleration of the vehicle. This relationship is nearly linear for a sufficiently large step-size. The model structure is demonstrated in figure 7. An example of the model output is shown in Figure 8, for various dynamics.



(a) Vertical motion dynamics (b) Roll dynamics (c) Pitch dynamics (d) Yaw dynamics

Figure 8. Reduced dynamics models and curve interpolation

This model can be inverted by using polynomial interpolation of the RBF net output and then solving it for the desired acceleration output. An array of virtual input samples $[v_0, v_1, \ldots, v_N]$ is supplied as input to the system and the acceleration outputs $\bar{y} = [\ddot{x}_{m,0}, \ddot{x}_{m,1}, \ldots, \ddot{x}_{m,N}]$ are recorded. A local polynomial curve approximation is generated every time a new virtual control input is requested, based on supplied target $\ddot{x}_d$. This curve can be expressed as

$$\ddot{\bar{x}}(v) = g(v) \cdot \theta, \tag{40}$$

where the parameter vector θ describes the interpolated polynomial that follows the shape of the neural net. In essence, the model itself represents the trim state voltage input for any given virtual input value, denoted by v_{input}. The parameter vector θ is modified by adding the desired acceleration $\ddot{x}_d$ to its first term, which describes the polynomial bias to reverse this model. Given a parameter vector θ that approximates the reduced model

$$\theta = \begin{bmatrix} p_0 & p_1 & \ldots & p_N \end{bmatrix}^T, \tag{41}$$

the adjusted parameter vector θ^* becomes

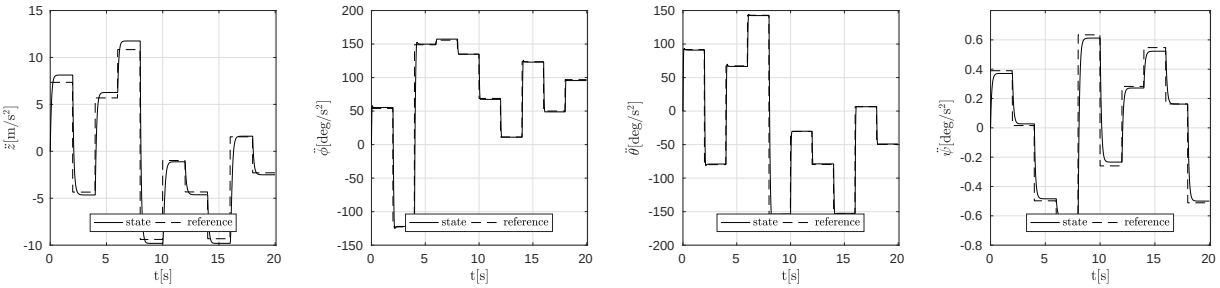
$$\theta^* = \begin{bmatrix} p_0 + \ddot{x}_d & p_1 & \ldots & p_N \end{bmatrix}^T. \tag{42}$$

The roots of the resulting system are found by calculating the eigenvalues of the companion matrix A^*

$$A^* = \begin{bmatrix} -\frac{p_1}{p_0+\ddot{x}_d} & -\frac{p_2}{p_0+\ddot{x}_d} & \cdots & -\frac{p_N}{p_0+\ddot{x}_d} \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{43}$$

$$\bar{v} = eig(A^*) \tag{44}$$

These roots $\bar{v}$ correspond to trim state values of the virtual inputs for a given desired target acceleration $\ddot{x}_d$. Due to additional dynamics taking place if this value was applied directly, the system might start moving towards the desired reference, but it might reach a local equilibrium before it reaches the goal. In other words, the controller would have a severe undershoot. To account for this discrepancy a small proportional gain is applied. The bottom part of Figure 7 illustrates the full inversion controller scheme. Figure 9 shows the resulting performance of the inverse model state controller is for various dynamic modes. Note that the transitions happen very quickly relative to the timescale, so for the reduced dynamics it's safe to assume that these changes take place instantaneously, given a large enough time step.



(a) Vertical motion dynamics (b) Roll dynamics (c) Pitch dynamics (d) Yaw dynamics

Figure 9. Dynamics inversion controller reference tracking

VI. Controller parameters

The controller has several initialization parameters that have to be set in advance: neural net generalization function parameters, such as the input states, the number of neurons per-dimension, and any auxiliary states that might have an effect on the dynamics that have to be controlled. Also, there are $\mathcal{Q}$-learning trial and value function parameters: duration of one episode, value function update rate α, discount parameter γ, trace decay rate λ, and the period during which the selected action is applied.

The trial length, discount rate, and action duration length are the parameters that have a major effect on the resulting policy: control dynamics such as pitch and roll have a much shorter response time, compared to yaw control for example. This difference in response time means that the time horizon over which an episode is executed should be shorter, and the inputs have to be adjusted more rapidly.

Another learning parameter that has to be set in advance is the reward function. There are two basic ways to do it: one is to assign a fixed penalty at each time step, until the goal is reached and terminal reward is awarded. This method is difficult to apply if the goal is not just some threshold to be reached, but a reference to be tracked over a period. A continuous state system may never reach the goal state exactly; there is always some small error present. Therefore, assigning an actual goal condition is difficult. Another way is to use a reward function that changes depending on the current system state. There are various ways to define it. The way it is assigned can influence the learning process during the exploration stage, as well as the overall behavior of learned policy. For example: if the reward is always positive, then the controller might give preference to actions that had been explored previously, despite the possibility that some of the un-tried actions might lead to higher value. Likewise, assigning negative rewards results in preference given to actions that had not been explored previously, despite the fact that the optimum action already had been tried. In effect, this works the same as pre-setting the value function. Another factor when designing a reward function where several states are combined is the maintenance of balance between different states.

For example, a reward function that assigns a lower penalty for smaller state error but does not take state derivative into account might result in high overshoot. On the other hand, a reward function that assigns a high penalty for converging "too fast" might also lead to a controller that has slower response time or unsatisfactory steady-state performance.

A. Policy training

It had been shown that the computation time for an RL algorithm could grow exponentially[20] depending on the number of states. However, it was also demonstrated that using more efficient exploration techniques this time could be reduced to rise polynomially,[21] so exploiting efficient exploration techniques is paramount when dealing with a highly complex model.

The training procedure begins with the preliminary generation of several potential policy solutions. The policies are initialized with the weights of the neural net used to store the $\mathcal{Q}$-function initialized to 0. Several training episodes are executed. The training progresses for as long as consecutive updates produce an improvement in evaluation score. The policies are evaluated by performing a test run using the simplified model while the controller is enabled and recording the average cumulative reward. The test signal consists of a mix of step and sinusoidal inputs. Step input allows evaluating steady state performance of the controller, while the sinusoidal signal is designed to evaluate the response of the controller to a moving reference signal.

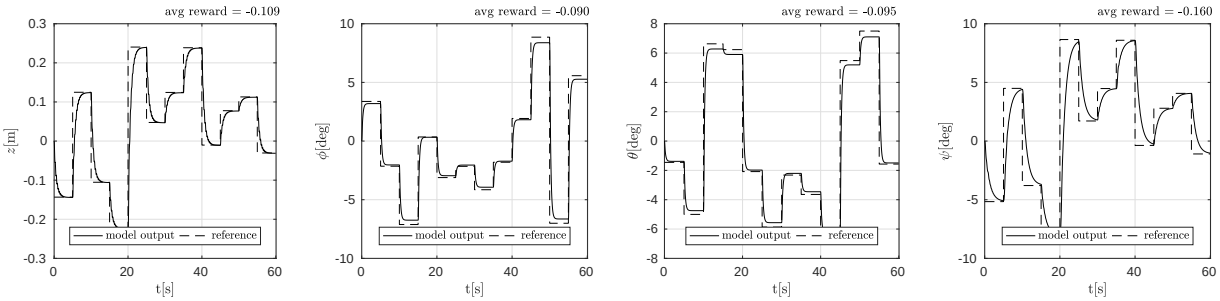
There are three training methods used to train the policy: random state sweep, grid-based state sweep and variable signal response. In a random state sweep, the initial state is placed randomly, close to the final goal state. As the learning progresses, the initial state is placed further and further from the goal. This approach allows to generate a policy valid near the goal region quickly and then refine it by moving the initial position further away. During the grid-based search, the initial starting points sweep the entire state-space of the controlled system. This strategy is designed to cover the entire state-space and to cover the states that might not have been visited during a randomized search. Variable signal training is done by letting the simulation run continuously and varying the reference signal(goal). Continuously adjusting the goal state allows simulating the actual conditions of the system in operation. During the training stage, the strategy is selected at random.

After generating several fully trained policies, the best one is selected. There is a considerable spread between various policy scores. The policy quality starts to decrease after a few initial test runs. Therefore it is important to be able to detect it and prevent it from getting worse as the learning continues on the best policy.

VII. Simulation results

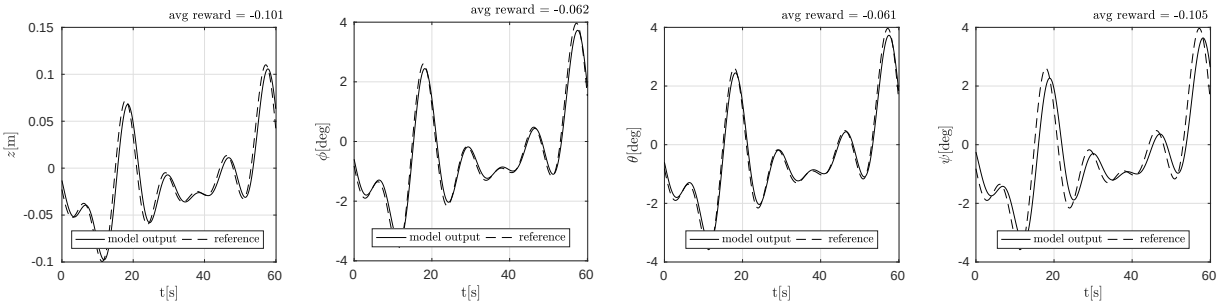
To evaluate the performance of the trained system and compare it against the PD two types of signals are used: a variable step input and a sinusoidal pseudo-random input. The controllers are evaluated in isolation - e.g., only one controller is activated at the time for each dynamic mode.

A PD controller is implemented to serve as a benchmark for the $\mathcal{Q}$-learning controller performance. The PD gains are tuned using PSO optimization. Time response of the resulting controller is shown in figures 10 - 11 for various dynamic modes.



(a) Vertical motion controller (b) Roll controller (c) Pitch controller (d) Yaw controller

Figure 10. PID controller square wave signal response

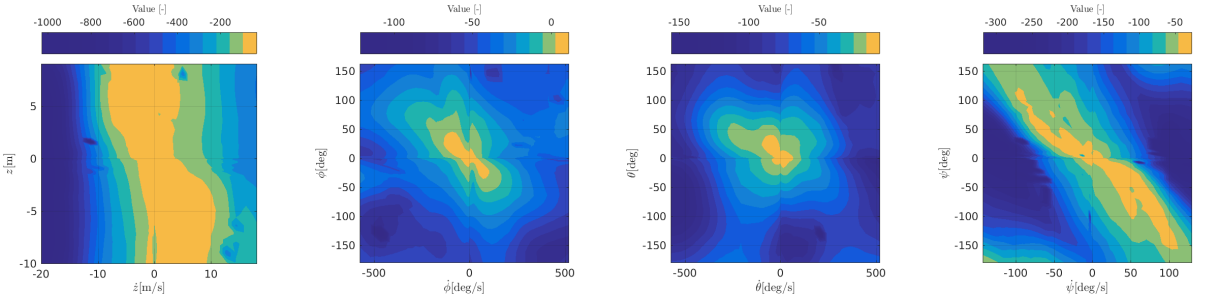


(a) Vertical motion controller (b) Roll controller (c) Pitch controller (d) Yaw controller

Figure 11. PID controller sine wave signal response

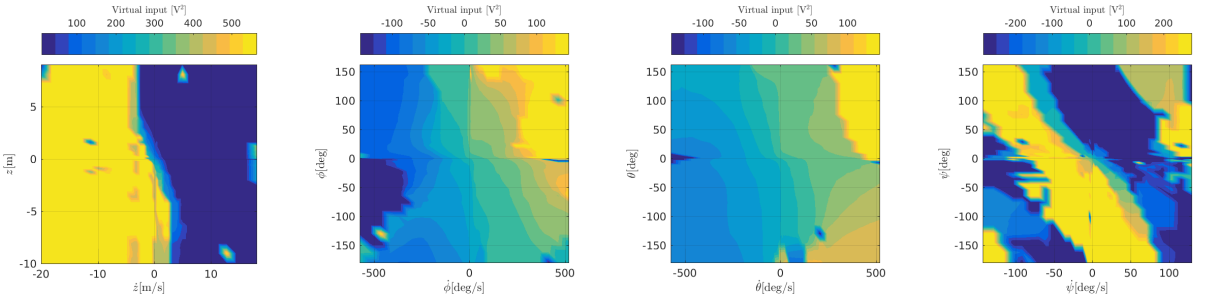
The PID controller performance is scored by calculating the average of all rewards received through the test trial. During these trials the full model of the process is used(with motor dynamics included). The resulting score allows to make an objective comparison against the performance of a reinforcement learning based approach.

The $\mathcal{Q}$-learning controller is trained for four dynamic modes: altitude, pitch, roll, and yaw control. The resulting trained value functions and the action maps are shown in Figures 12 and 13.



(a) Altitude controller (b) Roll controller (c) Pitch controller (d) Yaw controller

Figure 12. RL controller value function



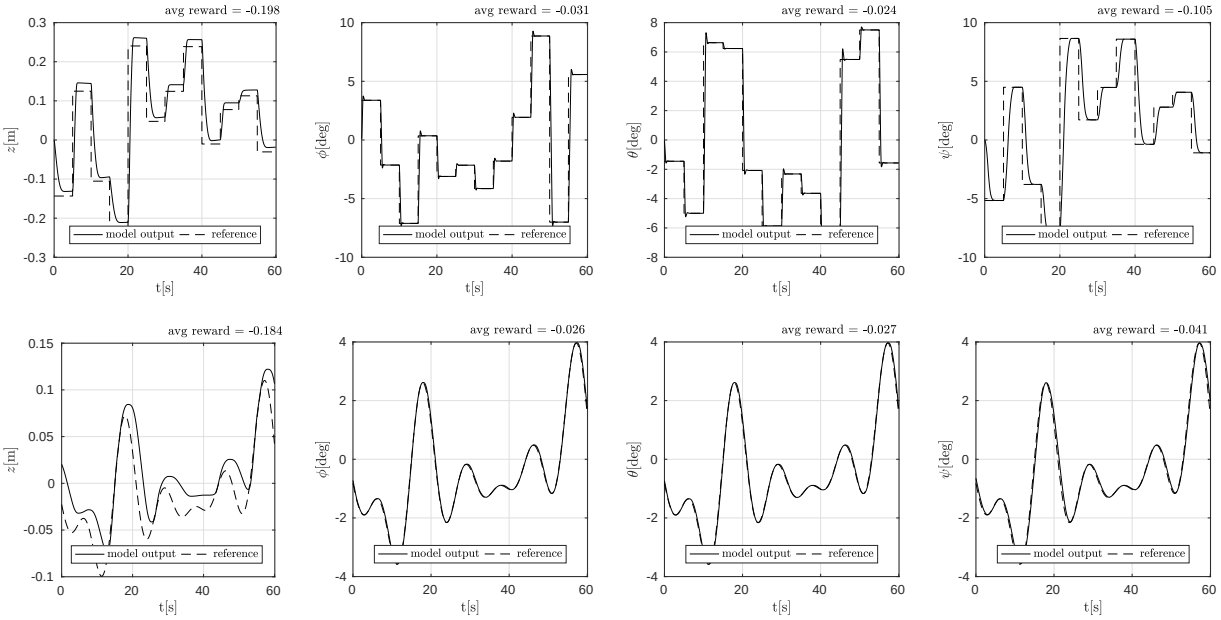
(a) Altitude controller (b) Roll controller (c) Pitch controller (d) Yaw controller

Figure 13. RL controller action map

The approximated value function plot shows that the value is much higher near the goal state, and it diminishes further away from the goal. This behaviour is expected of $\mathcal{Q}$-learning. The pitch, roll and yaw controller value functions are symmetrical, whereas the altitude controller value function exhibits some bias depending on whether the drone is moving upwards or downwards. This bias can be attributed to the fact that the actuation limits are not symmetrical: the lower limit of the climb rate cannot exceed gravitational acceleration(free fall), while the upper limit is capped by the available thrust power.

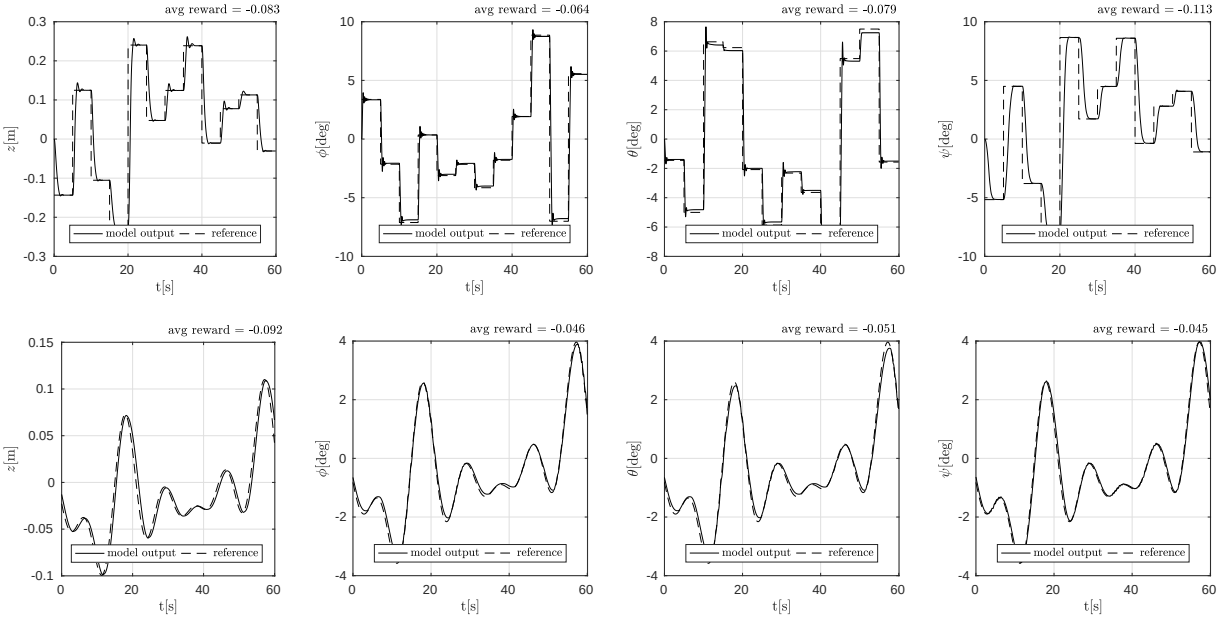
Another interesting feature is the behaviour of the Yaw controller. The action map looks like a sequence of bands. This nonlinearity can be explained by the fact that the yaw controller has the least actuation power, reacting very slowly and the states themselves are cyclic, meaning that as the aircraft rotates, it can flip from negative to positive reference offset. So for certain combinations of high offset from the reference state and high yaw rate, the controller tends to flip the direction of the applied control, anticipating future state. The altitude and yaw controllers react very sharply compared to pitch and roll controllers. This effect can also be attributed to the actuation power available to different controllers. The roll and the pitch controllers tend to "feather" the applied virtual input, whereas the altitude and yaw controllers quickly switch between the minimum and the maximum available actuation forces.

Due to nature of the algorithm there are two possible ways to evaluate the resulting policy: using the reduced model of the vehicle that neglects some dynamic aspects or a full model that introduces some delay in response and fits the system more closely. Time series of these responses are shown in Figures 14-15 for the reduced and the full model.



(e) Altitude controller (f) Roll controller (g) Pitch controller (h) Yaw controller

Figure 14. RL controller time response, reduced model



(e) Altitude controller (f) Roll controller (g) Pitch controller (h) Yaw controller

Figure 15. RL controller time response, full model

There is a difference between reduced and full models of the system. The full model exhibits a higher overshoot and some steady-state error, compared to the reduced model. As a consequence, the average trial score is lower for the full model, compared to reduced model in most cases. A complete list of test run scores in shown in tables 3 - 6.

	Score full test run	Score square input	Score sinusoidal input
RL controller, reduced model	-0.194	-0.198	-0.184
RL controller, full model	-0.086	-0.083	-0.092
PID controller	-0.104	-0.109	-0.101

Table 3. Test run score comparison, Altitude control

	Score full test run	Score square input	Score sinusoidal input
RL controller, reduced model	-0.030	-0.031	-0.026
RL controller, full model	-0.054	-0.064	-0.046
PID controller	-0.079	-0.090	-0.062

Table 4. Test run score comparison, Roll control

	Score full test run	Score square input	Score sinusoidal input
RL controller, reduced model	-0.026	-0.024	-0.027
RL controller, full model	-0.064	-0.079	-0.051
PID controller	-0.078	-0.095	-0.061

Table 5. Test run score comparison, Pitch control

The reinforcement learning controller score is higher than the score of PID controller, optimized for the same objective function. This indicates that reinforcement-learning controller performance is on par with the PID controller, even slightly better, given comparison based on objective function alone.

	Score full test run	Score square input	Score sinusoidal input
RL controller, reduced model	-0.074	-0.105	-0.041
RL controller, full model	-0.079	-0.113	-0.045
PID controller	-0.130	-0.160	-0.105

Table 6. Test run score comparison, Yaw control

VIII. Conclusions

A practical continuous state, continuous action $\mathcal{Q}$-learning controller framework has been described and tested using a model of a multicopter. This work demonstrates that RL methodology can be applied to inner-loop control of UAVs. The described approach combines model identification and offline learning using a reduced order model of the plant. The model can be adjusted online to achieve a more accurate estimation of the plant dynamics. An optimized hashed neural network algorithm used to store the Q-function values allows to optimize the computational load of the algorithm, making it suitable for online applications. The performance of the algorithm was validated and compared against that of a conventional proportional-derivative controller and was found to exceed it. The direction of future research would include testing the resulting algorithm in experimental setting that involves a physical system(a multicopter drone), extending the control scheme to include outer loop control or applying the developed methodology to other classes of robotic systems with continuous states and actions.

References

[1]Kaelbling, L. P., Littman, M. L., and Moore, A. W., "Reinforcement Learning : A Survey," *Journal of Artificial Intelligence Research*, Vol. 4, 1996, pp. 237–285.

[2]Szepesvári, C., "Algorithms for Reinforcement Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 4, No. 1, 2010, pp. 1–103.

[3]Sutton, R. S. and Barto, A. G., "Reinforcement learning: an introduction." *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, Vol. 9, 1998, pp. 1054.

[4]Narendra, K. S. and Parthasarathy, K., "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, Vol. 1, No. 1, 1990, pp. 4–27.

[5]Gaskett, C., Wettergreen, D., and Zelinsky, A., "Q-learning in continuous state and action spaces," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 1747, Springer Verlag, 1999, pp. 417–428.

[6]Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E., "Autonomous inverted helicopter flight via reinforcement earning," *Springer Tracts in Advanced Robotics*, Vol. 21, 2006, pp. 363–372.

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

[8]Zhang, B., Mao, Z., Liu, W., and Liu, J., "Geometric Reinforcement Learning for Path Planning of UAVs," *Journal of Intelligent and Robotic Systems: Theory and Applications*, 2013, pp. 1–19.

[9]Gaskett, C., "Q-learning for robot control," *Dspace.Anu.Edu.Au*, Vol. 1, 2008.

[10]Sutton, R., "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in neural information processing systems*, 1996, pp. 1038–1044.

[11]McCulloch, W. S. and Pitts, W., "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, Vol. 5, No. 4, 1943, pp. 115–133.

[12]Pitts, W. and McCulloch, W. S., "How we know universals the perception of auditory and visual forms," *The Bulletin of Mathematical Biophysics*, Vol. 9, 1947, pp. 127–147.

[13]Hagan, M. T. and Menhaj, M. B., "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, Vol. 5, No. 6, 1994, pp. 989–993.

[14]Albus, J. S., "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)," *Journal of Dynamic Systems, Measurement, and Control*, Vol. 97, No. 3, 1975, pp. 220.

[15]Bouabdallah, S., Noth, A., Siegwart, R., and Siegwan, R., "PID vs LQ control techniques applied to an indoor micro quadrotor," *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, Vol. 3, 2004, pp. 2451–2456.

[16]Mohammadi, M., Shahri, A. M., and Boroujeni, Z., "Modeling and Adaptive Tracking Control of a Quadrotor UAV," *International Journal of Intelligent Mechatronics and Robotics*, Vol. 2, No. 4, jan 2012, pp. 58–81.

[17]Nagaraj, B. and Murugananth, N., "A comparative study of PID controller tuning using GA, EP, PSO and ACO," *Communication Control and Computing Technologies (ICCCCT), 2010 IEEE International Conference on*, 2010, pp. 305–313.

[18]Gaing, Z.-L. L., "A Particle Swarm Optimization Approach for Optimum Design of PID Controller in AVR System," *IEEE Transactions on Energy Conversion*, Vol. 19, No. 2, 2004, pp. 384–391.

[19]Chao, O. and Weixing, L., "Comparison between PSO and GA for parameters optimization of PID controller," *2006 IEEE International Conference on Mechatronics and Automation, ICMA 2006*, Vol. 2006, 2006, pp. 2471–2475.

[20]Whitehead, S. D., "A Complexity Analysis of Cooperative Mechanisms in Reinforcement Learning," *AAAI-91 Proceedings*, 1991, pp. 607–613.

[21]Carroll, J., Peterson, T., and Owens, N., "Memory-guided exploration in reinforcement learning," *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, Vol. 2, No. January, 2001, pp. 1–44.

Chapter 3

Reinforcement learning

Reinforcement learning and intelligent control are used increasingly in various control applications. One of the issues associated with it is how to apply those controllers in safety-critical situations. In this case, the system must be *taught* to behave safely.

The main challenge is how to recognize potentially unsafe situations in advance, with some degree of certainty. If that's the case, using a model of the plant allows achieving some forecasting capability for the system. With more training data, however, it is possible to build a control logic that would avoid such situations altogether, only making decisions that are guaranteed to be safe.

Reinforcement learning can deal with the nonlinearities that occur in most real-life systems, and it is highly adaptable and adaptive. As such, it had been studied and applied to a wide range of problems in a variety of different fields. In reinforcement learning the system consists of an *agent*, acting in some *environment*. For example, an agent could be an autonomous vehicle, while the environment is the world in which it acts. The learning process itself consists of the actor taking different actions to *explore* the system and to *exploit* it by using the knowledge that it already has.

After each transition from agent state s to s' the agent receives some reward r. If successful, the agent receives a higher reward, and if the action leads to an undesirable state, then the magnitude of the reward is lower. This reinforcement takes the shape of a certain *reward function*, determined beforehand. In autonomous vehicle context, the rewards could be defined by smooth movements, quick response, the safety of the operation, etc. The purpose of an agent is to maximize this reward by taking optimal actions. By acting in a certain way, the actor follows a *policy* π, which represents the mapping from perceived states of the environment to actions to be taken (Sutton & Barto, 1998). While the reward function r determines the actions of an agent in a short run, influencing the adopted policy, a *value function* $V(s)$ is used to specify it's total reward, in the long term, determining the agent's behavior B. This behavior can be learned using a wide variety of different algorithms. This chapter introduces some of these algorithms and the underlying theory behind them.

Formally, the reinforcement model consists of:

- a set of environment states, $\mathcal{S}$, which can be discrete or continuous
- a set of actions $\mathcal{A}$, which can be discrete or continuous
- a set of reinforcement signals r, which can be static or varying, depending on agent state

3-1 Optimal Behavior Models

One of the specific attributes of using Reinforcement Learning techniques for control is that the agent can perceive some expected future rewards, depending on the state it's in and the course of actions that it follows. There are three basic models to represent this kind of behavior.

In the *finite-horizon* model the agent strives to optimize its expected cumulative reward r over h steps (Kaelbling, Littman, & Moore, 1996):

$$E\left(\sum_{t=0}^{h} r_t\right) \tag{3-1}$$

The r_t in this expression represents the reward, received at time t. There are two ways to use this reinforcement: one is to have a non-stationary policy, that would change over time. At the first step, the agent would take an *h-step optimal action*, the best action, given that there are h steps remaining for the actor to act. At the next step, it takes $(h-1)$-optimal action, until it finally takes 1-step optimal action and terminates. Another approach is to use *receding-horizon* control, in which the agent always takes h-step optimal actions. In real applications, the finite horizon approach is not always practical. Often it is not known, exactly how long will it take the agent to accomplish the task or for how long will it remain active.

The *infinite-horizon* model takes the long-run reward into account, but rewards received in the future are discounted, according to some discount factor γ ($0 \leq \gamma < 1$).

$$E\left(\sum_{t=0}^{h} \gamma^t r_t\right) \tag{3-2}$$

With the discount factor γ set to zero, the agent only learns a 1-step optimal policy, as it only considers the immediate reward. With the discount factor set to 1, the agent takes into account the cumulative reinforcement received over its entire lifetime. Mathematically, setting the discount factor γ to 1 can lead to divergence of expected reward sum, if an appropriate termination condition is not specified. The high value of γ can also slow down learning since the possible return of all future actions is considered. The long-run expected rewards might overshadow the most immediate reward. So it's hard to distinguish a policy which gains a significant amount of reinforcement initially, and the lower amount over the long run from a policy that generates a moderate amount of rewards throughout.

Another approach to optimality is the *average-reward* model. Using this criterion, the agent takes actions aimed to maximize the average reward over the long run.

$$\lim_{h\to\infty} E\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right) \tag{3-3}$$

This policy is referred to as a *gain optimal* policy. It is similar to the infinite-horizon model, as the discount factor γ is approaching 1, and suffers from the same issues, considering the balance between the short term and the long term rewards. It is possible to avoid that by using a *bias optimal* model, where both the initial reinforcement and the long term average are taken into account.

3-2 Learning Performance Measurement

There are various criteria to evaluate learning and policy performance. Measuring it is important to select the right approach for the task. The two most important criteria are the optimality of learned policy and the speed of convergence. There are some algorithms that are proven to converge to an optimal policy eventually, but sometimes it is not practical because it might take a lot of time to converge. Other, faster algorithms, might reach the plateau performance more quickly, but the resulting performance might be sub-optimal.

A good online reinforcement learning algorithm must satisfy two requirements. First of all, it must be capable of finding an optimal control solution quickly by exploring the environment. Second, as the solution is refined, it must converge to an optimum solution, while avoiding being stuck in a local optimum (Busoniu, Babuška, De Schutter, & Ernst, 2010).

3-3 Reinforcement learning methodologies

Broadly speaking, the reinforcement learning methods can be classified into two categories: model-based and model-free. In a purely model-based approach, it is assumed that an agent has an explicit model of the environment and that its state can be fully observed. In a strictly model-free approach, the agent learns only from experience, without a model or knowledge of the environment dynamics.

Dynamic Programming (DP) algorithms belong to the class of model-based approaches (Cybenko, 1998). DP methods operate on two principles: policy iteration and value iteration. Policy iteration means typically iterative computation of the value function for a given policy. Value iteration involves calculation of an improved policy, based on value function. Classic DP methods operate by performing sweeps through the state set, performing a backup operation in each state and updating corresponding values.

DP methods have several significant drawbacks that make them unsuitable for real-world applications. They require sweeps of an entire state set, which can be prohibitively expensive if the state is too large. They are also computationally expensive. Some of it can be mitigated by using asynchronous DP algorithms, which update the values with whatever values are available at the time and don't require exhaustive sweeps. These are preferred for problems with large state spaces. They also make it easier to mix computation and real-time data. A

DP algorithm can be run at the same time as the agent acts. Nevertheless, it always requires a model to assign the rewards correctly.

Other methods are less explicitly dependent on an existing model, and that can learn from experience, on-line or simulated. Monte-Carlo methods can utilize this principle (Sutton & Barto, 1998). In Monte Carlo methods the experience is divided into episodes. Once the episode is completed, the value estimates and policies are changed. It is different from DP in a sense that the estimates for each state are independent of all the other states. This gives Monte Carlo an advantage of reduced computational expense and makes it more attractive when values of only a subset of the states are required. If a model is available, the state values alone are sufficient to design a policy. If it isn't, however, the value must be estimated. This estimation can be done by averaging the returns as the number of visits to each state-action pair reaches infinity. Monte Carlo methods also have some issues, such as the requirement to maintain sufficient exploration. It is not enough to take optimal actions. There could be better alternative actions, but unless they are visited, they remain unknown.

Temporal Difference (TD) learning can learn directly from experience like in case with Monte Carlo methods. In both cases, the value estimate is updated based on the outcome from visiting some nonterminal state. But unlike the case with Monte Carlo, it is not necessary to wait until an episode is complete to update the value function. TD methods can immediately form a new target. As such, TD methods can be naturally implemented in an on-line fashion which gives them an advantage. Another advantage is that TD performs faster than Monte Carlo methods. And unlike the case of DP, this can be done without any knowledge of the model. $\mathcal{Q}$-learning and Actor-Critic methods are subsets of TD, where the former is more suitable for discrete systems and the latter is designed to deal with continuous models. $\mathcal{Q}$-learning has an advantage over actor-critic method because it is exploration-insensitive, i.e., it can learn without following the current policy. However, it has to be adapted to deal with continuous states and continuous inputs, which constitute integral elements of most real-world applications. This presents some difficulty in applying $\mathcal{Q}$-learning to problems with smoothly varying states and smoothly varying actions (Gaskett, 2008).

In a sense, the methods described are not mutually exclusive, but they provide a set of tools that can be applied to solve a particular problem. Monte Carlo and Temporal Difference methods are of most interest when considering the objective of designing an intelligent UAV controller that can act with minimum knowledge about the model and learn in real time. They are less computationally intensive than DP do not require a complete model that is known a priori. They do, however, require some knowledge of system dynamics that can be found by building a model.

3-4 Delayed reward

In addition to immediate rewards, a reinforcement learning agent must be able to take future rewards into account. Therefore it must be able to learn from delayed reinforcement. Typically the agent would progress throughout it's learning process, starting off with small rewards, and then receiving larger rewards as it gets closer to its goal. And it must learn whichever actions are appropriate at any instance, based on reinforcement that can be received at any time in the future.

3-5 Markov decision process

Delayed reinforcement learning problems can be modeled as *Markov decision processes*(MDP). An MDP consists of

- a set of states $\mathcal{S}$
- a set of actions $\mathcal{A}$
- a reward function $R : \mathcal{S} \times \mathcal{A} \rightarrow \Re$
- a state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, where Π maps the states to probabilities. The probability of transition from state s to state s' given action a.

The state transition function specifies the transition from one state to another, following an action that was taken by the agent. The instant reward received following an action. The model has *Markov* property if the state transitions are independent of any previous environment states or agent actions.

3-6 Policy search using a model

For an infinite-horizon discounted model there exists an optimal deterministic stationary policy. The optimal *value* is the expected infinite discounted sum of reward at any given state. The value is described as

$$V^*(s) = \max_{\pi} E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \tag{3-4}$$

Here π represents the decision policy. The value is unique, and it can be described as the solution to the simultaneous equations

$$V^*(s) = \max_{a}\left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')V^*(s')\right), \forall s \in \mathcal{S} \tag{3-5}$$

This value includes the expected instantaneous reward R and the expected discounted value over the long run, when using the best available actions a, following the policy π:

$$\pi^*(s) = \arg\max_{a}\left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')V^*(s')\right) \tag{3-6}$$

3-7 Value iteration

In order to predict the outcome of a Markov chain, an incremental form of value iteration could be used:

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha(r + \gamma V(s')) \tag{3-7}$$

This basic operation is used in most temporal difference algorithms. For the problem of prediction on a deterministic Markov chain, the goal can be defined as satisfying the Bellman Equation, where each transition from state s to state $'s$ yields an immediate reward r:

$$V(s) = (r + \gamma V(s')) \tag{3-8}$$

For a given value function V the *Bellman residual* is defined as the difference between two sides of the Bellman equation. The *mean squared Bellman residual* for an MDP is defined as:

$$E = \frac{1}{n}\sum_{s}[(r + \gamma V(s')) - V(s)]^2 \tag{3-9}$$

Where n represents the number of states.

3-8 $\mathcal{Q}$-learning

One of the most important reinforcement learning algorithms available today is $\mathcal{Q}$-learning(Sutton & Barto, 1998). It is an off-policy TD control algorithm. Off-policy means that it learns the action-values that are not necessarily on the policy that it is following. The $\mathcal{Q}$-learning is using a $\mathcal{Q}$ function to learn the optimal policy. The expected discounted reinforcement of taking action a at state s is represented as $\mathcal{Q}^*(s,a)$.

$$\mathcal{Q}^*(s) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') \max_{a'} \mathcal{Q}^*(s',a') \tag{3-10}$$

This function is related to value as $V^*(s) = \max_a \mathcal{Q}^*(s,a)$. Then the optimal policy is described as $\pi^*(s) \arg\max_a \mathcal{Q}^*(s,a)$. One-step $\mathcal{Q}$-learning is defined by the following value update rule:

$$\mathcal{Q}(s,a) \leftarrow \mathcal{Q}(s,a) + \alpha(r + \gamma \max_{a'} \mathcal{Q}(s',a') - \mathcal{Q}(s,a)) \tag{3-11}$$

In this equation, $\mathcal{Q}$ stands for action-value function, updated at every step at a learning rate α. The update parameters are the reward r and the value of $\mathcal{Q}$ at the next step, corresponding to the maximum possible reward for a certain action a. Factor γ is the discount rate; it represents the eligibility of the value for a change. It makes rewards earned earlier more valuable than the ones received later. For state s at a discrete time k eligibility trace is denoted as $e_k(s) \in \mathbb{R}^+$. At each time step in a trial, the eligibility traces e decays by a factor

$\gamma\lambda$, where λ is the trace-decay parameter(Sutton & Barto, 1998). The eligibility trace for a state that just had been visited can be incremented by 1.

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1}(s) & \text{if } s \neq s_k \\ \gamma\lambda e_{k-q}(s) + 1 & \text{if } s = s_k \end{cases} \tag{3-12}$$

This method is referred to as *accumulating traces*. Anoter method of updating traces is follows:

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1}(s) & \text{if } s \neq s_k \\ 1 & \text{if } s = s_k \end{cases} \tag{3-13}$$

This method is called *replacing traces*. The accumulating traces is known to suffer from convergence issues, replacing traces method is thus mostly used.

The general procedure for $\mathcal{Q}$-learning is shown as follows:

Data: Initialize $\mathcal{Q}(s, a)$ arbitrarily
for *each episode* **do**
  Initialize s
  for *each step* **do**
    Choose action a from s using policy derived from $\mathcal{Q}$
    take action a, observe reward r, new state s'
    $\mathcal{Q}(s_k, a_k) \leftarrow \mathcal{Q}(s_k, a_k) + \alpha[r_{k+1} + \gamma \max_a \mathcal{Q}(s_{k+1}, a) - \mathcal{Q}(s_k, a_k)]$
    $s \leftarrow s'$
    if *goal is reached* **then**
      break
    end
  end
end

3-9 Sarsa

There are numerous variations to $\mathcal{Q}$-learning algorithm, aiming at improving convergence and optimality characteristics, and trading off exploration and exploitation during the learning process. Some of them will be discussed in this section.

Sarsa algorithm is an on-policy variation of $\mathcal{Q}$-learning (Kaelbling et al., 1996). An action-value function has to be learned rather than an state-value function. For an on-policy method $\mathcal{Q}^\pi(s, a)$ must be estimated. This can be done using a nearly identical kind of update as in the general $\mathcal{Q}$-learning. Sarsa learning update rule is defined as follows:

$$\mathcal{Q}(s, a) \leftarrow \mathcal{Q}(s, a) + \alpha(r + \gamma\mathcal{Q}(s', a') - \mathcal{Q}(s, a)) \tag{3-14}$$

The difference with the $\mathcal{Q}$-learning update, described earlier is that rather than the optimal $\mathcal{Q}$-value $\max_{a'} \mathcal{Q}(s', a')$ the value of the action taken $\mathcal{Q}(s', a')$ is used during the update. The full update algorithm is shown as follows:

Data: Initialize $\mathcal{Q}(s, a)$ arbitrarily
for *each episode* **do**
  Initialize s
  Choose action a from s using policy derived from $\mathcal{Q}$ (e.g. ε-greedy)
  for *each step* **do**
    Choose action a from s using policy derived from $\mathcal{Q}$ (e.g. ε-greedy)
    take action a, observe reward r, new state s'
    $\mathcal{Q}(s, a) \leftarrow \mathcal{Q}(s, a) + \alpha[r + \gamma\mathcal{Q}(s', a') - \mathcal{Q}(s, a)]$
    $s \leftarrow s'$
    if *goal is reached* **then**
      break
    end
  end
end

3-10 Dyna-$\mathcal{Q}$ learning

In Dyna approach, the agent experience is used to build a model, while simultaneously using this model to learn (Neumann, 2005). The idea behind Dyna is to preserve the state transition experience tuples $\langle s, a, s', r\rangle$ and use these to update models $\hat{R}(s, a)$ and $\hat{T}(s, a)$. The policy updates are then performed as follows, for each experience tuple:

- The model is updated using the information about transition from s to s', following an action a.
- The policy is updated, according to value-iteration rule for the $\mathcal{Q}$-values:

$$\mathcal{Q}(s, a) \leftarrow \hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s, a, s') \max_{a'} \mathcal{Q}(s', a') \tag{3-15}$$

- Additional k updates are performed, after choosing k state-action pairs at random

$$\mathcal{Q}(s_k, a_k) \leftarrow \hat{R}(s_k, a_k) + \gamma \sum_{s'} \hat{T}(s_k, a_k, s') \max_{a'} \mathcal{Q}(s', a') \tag{3-16}$$

Dyna algorithm increases computational complexity of the updates, requiring k further iterations of the update rule per each update step. However, it also tends to converge after fewer update steps, compared to traditional $\mathcal{Q}$-learning, requiring fewer experience steps.

3-11 Double $\mathcal{Q}$-learning

In some environments $\mathcal{Q}$-learning is performing poorly, due to overestimations of action values, that results from a positive bias that is introduced because $\mathcal{Q}$-learning uses maximum action value as an approximation for the maximum expected action value. Double $\mathcal{Q}$-learning algorithm (Hasselt, Group, & Wiskunde, 2010) is designed to mitigate these effects. In essence,

$\mathcal{Q}$-learning is using a single estimator to estimate the value of the next state: $\max_a \mathcal{Q}_t(s', a)$ is an estimate for $E\{\max_a \mathcal{Q}_t(s', a)\}$, which in turn approximates $\max_a E\{\mathcal{Q}_t(s', a)\}$. This expectation is averaging over all possible runs of the same experiment, and not as the expectation over the next state. The value of $\max_a \mathcal{Q}_t(s', a)$ is an unbiased sample with mean $E\{\max_a \mathcal{Q}_t(s', a)\}$. Because of this $\mathcal{Q}$-learning can suffer from large overestimations.

In double $\mathcal{Q}$-learning two $\mathcal{Q}$-functions are stored: $\mathcal{Q}^A$ and $\mathcal{Q}^B$. Each is updated with the value from another $\mathcal{Q}$-function for each step of the learning process. So, instead of using the value $\mathcal{Q}^A(s', a^*) = \max_a \mathcal{Q}^A(s', a)$ to update the function $\mathcal{Q}^A$, estimate $\mathcal{Q}^B(s', a*)$ is used instead. Since each of these functions is updated on the same problem, but with a different set of experience samples, this produces an unbiased estimate of the value of the action. The same logic is applied when updating the value of $\mathcal{Q}^B$, using $\mathcal{Q}^A$. In general, $E\{\mathcal{Q}^B(s', a*)\} \leq \max_a E\{\mathcal{Q}^A(s', a^*)\}$. The full algorithm is shown as follows:

Data: Initialize $\mathcal{Q}^A(s, a)$, $\mathcal{Q}^B(s, a)$ arbitrarily
for *each episode* **do**
  Initialize s
  for *each step* **do**
    Choose action a from s using policy derived from $\mathcal{Q}^A$ and $\mathcal{Q}^B$
    take action a, observe reward r, new state s'
    Choose (e.g. random) either UPDATE(A) or UPDATE(B)
    if *UPDATE(A)* **then**
      Define $a^* = \arg\max_a \mathcal{Q}^A(s', a)$
      $\mathcal{Q}^A(s, a) \leftarrow \mathcal{Q}^A(s, a) + \alpha[r + \gamma \mathcal{Q}^B(s', a^*) - \mathcal{Q}^A(s, a)]$
    end
    else if *UPDATE(B)* **then**
      Define $b^* = \arg\max_a \mathcal{Q}^B(s', a)$
      $\mathcal{Q}^B(s, a) \leftarrow \mathcal{Q}^B(s, a) + \alpha[r + \gamma \mathcal{Q}^B(s', b^*) - \mathcal{Q}^B(s, a)]$
    end
    $s \leftarrow s'$
    if *goal is reached* **then**
      break
    end
  end
end

3-12 Advantage Learning

Advantage learning is a reinforcement learning algorithm in which two types of information are stored. For each state s the value $V(s)$, representing the estimate of the total discounted return at that state, and the value $\mathcal{A}(s, a)$, representing the amount by which the total discounted reinforcement could be increased by performing the action a (Harmon & Baird III, 1996). The optimal value function $V^*(s)$ represents the actual value of each state, while the optimal advantage function $\mathcal{A}^*(s, a)$ is converging towards 0 for optimal actions a, and remains negative if action a is sub-optimal. Advantage learning tends to learn faster than $\mathcal{Q}$-learning, especially for continuous-time problems.

The optimal advantage function $\mathcal{A}^*(s, a)$ and the value function are related as:

$$V^*(s) = \max_a \mathcal{A}^*(s,a)$$

Where the advantage for each state s and action a is defined as:

$$\mathcal{A}^*(s,a) = V^*(s) + \frac{r + \gamma^{\Delta t} V^*(s') - V^*(s)}{\Delta t K}$$

The following update rule for the advantage learning can be derived, similar to $\mathcal{Q}$-learning update:

$$A(s,a) \leftarrow A(s,a) + \alpha \left(\left(r + \gamma \max_a A(s',a) \right) \frac{1}{\Delta t K} + \left(1 - \frac{1}{\Delta t K} \right) \max_a A(s,a) - A(s,a) \right) \tag{3-17}$$

For continuous state and time types of problems, Advantage learning is found to converge to acceptable performance more often than pure $\mathcal{Q}$-learning (Gaskett, 2008).

3-13 Discussion

This chapter presented the main principles behind the reinforcement learning. Several RL methods were described and compared. $\mathcal{Q}$-learning algorithm was described. $\mathcal{Q}$-learning is an attractive temporal difference method that could be used to train a value function online, as well as from a training data set. Several variations of $\mathcal{Q}$-learning were discussed as well. In principle, $\mathcal{Q}$-learning was designed to solve discrete state and action problems. However, it can be adapted to deal with continues states and actions by using function approximation.

Chapter 4

Continuous state and action Q-learning

$\mathcal{Q}$-learning algorithms are commonly applied to problems with discrete sets of states and action. In those, the state-action spaces and rewards are encoded in tabular form, where each cell represents a combination of state and action. In reality, however, control problems where the states and actions are continuous are also widespread (Gaskett, Wettergreen, & Zelinsky, 1999). This presents a challenge in adapting the "classical" $\mathcal{Q}$-learning approach to deal with these types of problems. A common approach is to discretize the state of the world by splitting the state domain into discrete regions. This discretization approach, however, introduces some problems. For example if the state is coarsely discretized a *perceptual aliasing* problem occurs (Gaskett, 2008). It's hard to discretize world states without losing information. One of the solutions is to discretise the world more finely. This approach, however, increases the amount of memory required, introducing the curse of dimensionality. Fine discretization leads to an enormous state-action space.

These issues could be reduced by using *generalization*, e.g., using experience with a limited subset of the state space to produce an approximation over a larger subset (Sutton & Barto, 1998). This approach allows generalizing from previously experienced states to ones that hadn't been visited yet.

One type of generalization is *function approximation*, which in turn is an example of supervised learning. Most supervised learning methods seek to minimise the mean-squared-error (MSE) over distribution of inputs $P(s)$. Given target function value V^π the MSE for an approximation V_t using parameter $\bar{\theta}_t$ becomes -

$$MSE(\bar{\theta}_t) = \sum_{s \in \mathcal{S}} P(s)[V^\pi(s) - V_t(s)]^2 \tag{4-1}$$

A good strategy for estimating the parameter vector is to minimize this value on the observed examples by adjusting the parameter vector by a small amount at each data point, in the direction that would reduce the error.

$$\bar{\theta}_{t+1} = \bar{\theta}_t - \frac{1}{2}\alpha\nabla_{\bar{\theta}_t}[V^\pi(s_t) - V_t(s_t)]^2 = \bar{\theta}_t + \alpha[V^\pi(s_t) - V_t(s_t)]\nabla_{\dot{\theta}_t} \tag{4-2}$$

Where α is the step-size parameter and $\nabla_{\bar{\theta}} f(\bar{\theta}_t)$ is the derivative vector that represents the gradient of f with respect to $\bar{\theta}_t$

$$\left(\frac{\delta f(\bar{\theta}_t)}{\delta\theta_t(1)}, \frac{\delta f(\bar{\theta}_t)}{\delta\theta_t(2)} \cdots \frac{\delta f(\bar{\theta}_t)}{\delta\theta_t(n)}\right)^T \tag{4-3}$$

This gradient represents the direction in which the error falls most rapidly. Eventually the gradient-descent method converges to a local optimum. The general gradient-descent method for state-value approximation thus becomes:

$$\bar{\theta}_{t+1} = \bar{\theta}_t + \alpha[v_t - V_t(s_t)]\nabla_{\bar{\theta}_t} V_t(s_t) \tag{4-4}$$

With v_t being an unbiased estimate $E\{v_t\} = V^\pi(s_t)$ then $\bar{\theta}_t$ is guaranteed to converge to some local optimim.

A special case of gradient-descent approximation is in which the approximate function V_t is a linear function of the parameter vector, $\bar{\theta}_t$ (Sutton & Barto, 1998). In that case, there is some vector of features $\bar{\phi}_s$ such that

$$V_t(s) = \bar{\theta}_t^T\bar{\phi}_s = \sum_1^n \theta_t(i)\phi_s(i) \tag{4-5}$$

Linear learning methods could be very efficient approximators. When using them, the appropriate way to represent the state-space using features must be found.

Neural Networks (NN) can be used to model and control very complex systems with sufficient accuracy without complete knowledge of its inner composition. Neural networks themselves can be divided into two types: multilayer neural networks and recurrent networks. Despite some differences, they could be viewed in a unified fashion as part of a broader discipline (Narendra & Parthasarathy, 1990).

Another way to approximate a continuous function is using *coarse coding*. This is a technique where the state-space region is separated into overlapping regions, each representing a feature. In the binary case, if the state is situated inside of a feature, then the corresponding feature has a value of 1 and is said to be *present*. Otherwise, this feature is 0 and it is said to be *absent* (Sutton & Barto, 1998). The parameters that affect the discretization accuracy of this method are the size of the regions and number of overlapping regions.

4-1 Neural Networks

Neural Networks can be used to model and control very complex systems with sufficient accuracy without complete knowledge of its inner composition. Neural networks themselves can be divided into two types: multilayer neural networks and recurrent networks. Despite

some differences, they could be viewed in a unified fashion as part of a broader discipline (Narendra & Parthasarathy, 1990).

The concept of Neural Networks was first proposed by Warren McCulloch and Walter Pitts, who came up with a model of an artificial neuron, later dubbed *perceptron* (McCulloch & Pitts, 1943; Pitts & McCulloch, 1947).Their ideas were then criticized as being too limited in 1969, when a book 'Perceptrons' was published by authors Marvin Minsky and Seymour Papert (Nievergelt, 1969). One of their main critiques was that it was that it was impossible to simulate some logic gate functions with them. This had led to a decrease of interest in NN among the scientific community. However, before that, a Russian scientist Kolmogorov had demonstrated that these limitations could be overcome by using multilayer networks that in theory could be taught to approximate any function. In the following years, there was less research done regarding reinforcement learning. The situation had changed in the late 80s. Increasing ubiquity of personal computers and their growing power made reinforcement learning, in general, an attractive tool with many potential applications. In the following decades and up until now this research field was very active and, it is still growing and evolving. One of the fundamental problems in system identification when using NN is finding a suitable structure for the model, while parameter estimation in itself is a relatively straightforward process (Sjöberg et al., 1995).

The basic building unit of a neural network is an artificial neuron. The basic neuron consists of a weighted summer, the function of which is to combine one or several inputs. Each input is multiplied by some weight, which can be pre-determined, or adjusted as the learning process goes on. The combined and weighted inputs then pass through some nonlinear activation function. Various types of activation functions exist. It could be a simple threshold function, producing a unit output when some input threshold is passed. A radial-basis function that gradually increases or decreases, depending on the input value proximity to the center of the neuron. Ir a tangent-sigmoidal activation function, similar to a threshold function, but producing a smooth, varying output. Multiple neurons can be combined in layers, and layers themselves could be combined as well. A typical neural network would have one hidden layer, that consists of several nonlinear neurons and one output layer that often consists of a simple summation of weighted hidden layer outputs. The layouts of Radial Basis Function(RBF) and Feed-Forward(FF) neural nets are shown in figure 4-1.

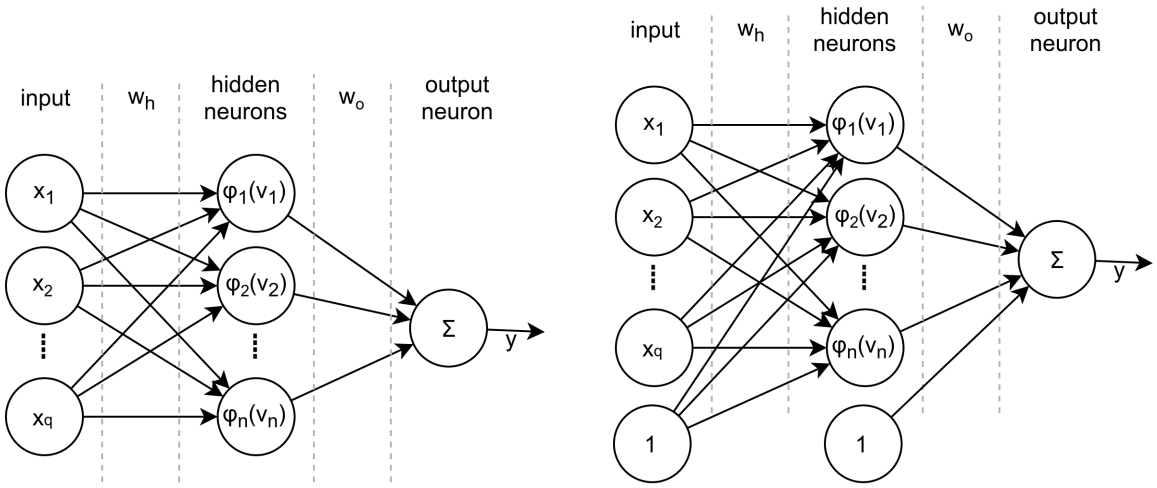
The activation function $\phi(v)$ can take various forms, depending on the intended application and desired characteristics of the neural net. Some of the typical ones are shown in figure 4-2.

There are a variety of training methods that apply to neural networks, e.g. linear regression methods, and gradient descent methods such as back-propagation, Levenberg-Marquardt, and others.

4-1-1 Linear regression training

Linear regression training is often applied to RBF neural networks. It is simple to use and computationally inexpensive. A typical linear regression model intended to train an RBF network can be defined as follows:

$$Y = A(x) \cdot \theta + \epsilon \tag{4-6}$$



(a) RBF neural net layout

(b) Feed-Forward neural net layout

Figure 4-1: Typical Neural Network layouts

In this equation Y denotes the recorded measurement vector $Y = [y_t^1, y_t^2, \cdots y_t^N]^T$, containing N datapoints, $A(x)$ is the regression matrix, containing hidden layer outputs for each recorded input x^k and θ is the vector, containing output layer weights $\theta = [w_{o,1}, w_{o,2}, \cdots, w_{o,n}]$ for each of n neurons in the hidden layer.

$$\nu_j(x_i) = \sum_{i=1}^{q} w_{h,i,j}(x_i - c_{i,j})^2 \tag{4-7}$$

$$\phi_j(\nu_j) = \mathrm{e}^{-\nu_i} \tag{4-8}$$

$$A(\phi) = \begin{bmatrix} \phi_1(\nu_1^1) & \phi_2(\nu_2^1) & \cdots & \phi_n(\nu_n^n) \\ \phi_1(\nu_1^2) & \phi_2(\nu_2^2) & \cdots & \phi_n(\nu_n^2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\nu_1^N) & \phi_2(\nu_2^N) & \cdots & \phi_n(\nu_n^N) \end{bmatrix} \tag{4-9}$$

where

q	number of input dimensions
n	number of hidden layer neurons
N	size of training data set
$\phi_i(v_i)$	RBF activation function
$c_{i,j}$	neuron centers
w_h	hidden layer input weights
w_o	output weights

In order to find an optimum set of parameters $\hat{\theta}$, the model error η has to be minimised:

$$\eta = Y - A(x) \cdot \theta \tag{4-10}$$

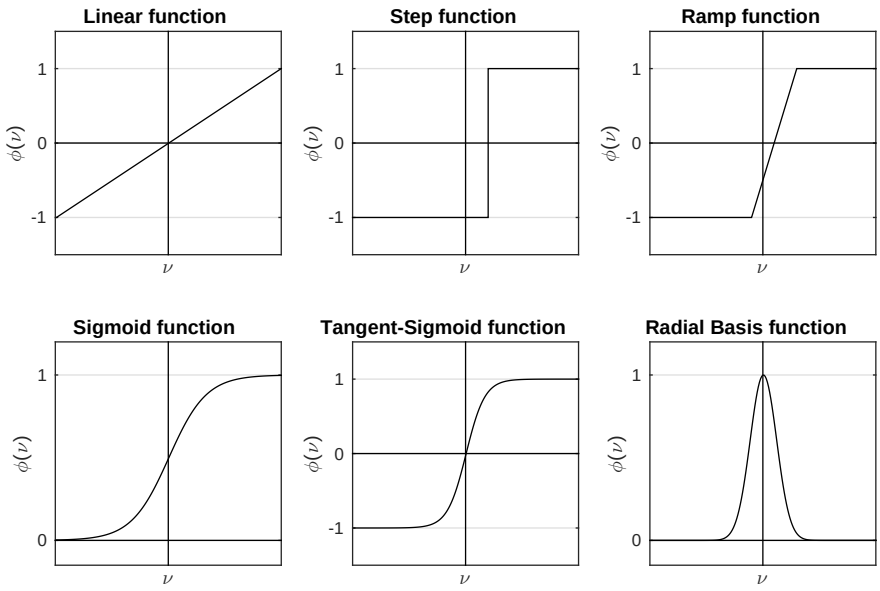


Figure 4-2: NN activation functions

In order to do so, a cost function J has to be defined. Then the optimal parameter vector $\hat{\theta}$ becomes:

$$\hat{\theta} = \arg\min J(Y - A(x) \cdot \theta) \tag{4-11}$$

Typically a quadratic error cost function is used, in order to avoid negative errors canceling out the positive ones. This function can be expressed as follows:

$$J(x, \theta) = (Y - A(x) \cdot \theta)^T (Y - A(x) \cdot \theta) \tag{4-12}$$

In order to minimise function $J(x, \theta)$ it is differentiated with respect to parameter vector θ:

$$\frac{\delta}{\delta\theta} J(x, \theta) = \frac{\delta}{\delta\theta} (Y - A(x) \cdot \theta)^T (Y - A(x) \cdot \theta) \tag{4-13}$$

The function can be differentiated using partial derivatives. Cost function is minimised when this derivative is zero.

$$\frac{\delta}{\delta\theta} J(x, \theta) = -2(Y - A(x) \cdot \theta)^T A(x) = 0 \tag{4-14}$$

Opening the brackets and re-arranging this equation leads to the following expression:

$$A^T(x) A(x) \cdot \hat{\theta} = A(x)^T Y \tag{4-15}$$

The parameter vector value $\hat{\theta}$ is the value that minimised the quadratic cost function $J(x, \theta)$. Thus, re-arranging equation (4-15):

$$\hat{\theta}_{OLS} = (A^T(x)A(x))^{-1}A^T(x)Y \tag{4-16}$$

This concludes the derivation of ordinary least squares method for updating of RBF network output weights. Note that only output weights are updated, while the input weights remain unchanged.

RBF neural net training demonstration

A simple two-dimensional function, shown in (4-17), was approximated to demonstrate the application of an RBF neural network. Figure 4-4(a) illustrates the output of the function.

$$f(\bar{x}) = \sin(2\pi x_1)\cos(2\pi x_2) \tag{4-17}$$

The output of the function is shown in figure 4-4(a). Identification and validation data sets were generating calculating the function output value for 500 randomly generated inputs to the function. Then several neural net configurations were created and trained using linear regression. Since only output weights were updated in this case, special care had to be taken when initiating input weights and neuron centers. There are various ways to initialise these: for example, they could be placed randomly or on a grid. In this case all network centers were put on a uniform grid within the state vector domain $x_1[0,1], x_2[0,1]$. The inputs weights were initialized as a function of the spacing between nearest center points on the grid $\Delta_{c,i}$ along i-direction:

$$w_{h,i,j}^{init} = 1/\Delta_{c,i} \tag{4-18}$$

This type of initialization allows covering the entire function domain uniformly.

The RMSE of the trained RBF neural net is shown in Figure 4-3, for various neural net configurations. It is evident that the identification error drops as the number of neurons increases, while the validation error levels off at some point. The outputs of some select networks are shown in Figure 4-4, along with the plot of the original function.

The plot shows that a network with 12 neurons is insufficient for approximating function features, as it demonstrates heavy aliasing. The network with 20 neurons already approximates the original function well, while a network with 110 neurons produces a surface nearly identical to original function with very low error levels.

Note that neural nets are relatively expensive to implement computationally. A network with q inputs and n hidden layer neurons requires equations (4-7) to be evaluated $q \times n$ times, and (4-8) to be assessed n times for each forward pass. This can be computationally taxing, especially considering that equation (4-8) is an exponential function, and exponential functions are much slower to execute than simple operations such as addition or multiplication. This computational overhead can be reduced by only considering the neurons activated at a time and approximating the exponential function using a polynomial or a lookup table with pre-calculated values.

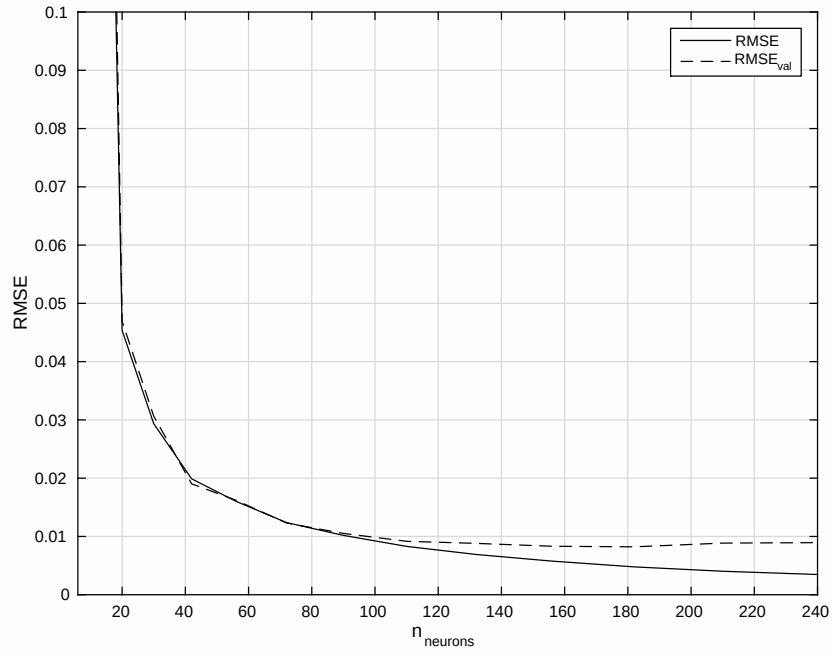


Figure 4-3: RMSE and memory size of various RBF neural net configurations

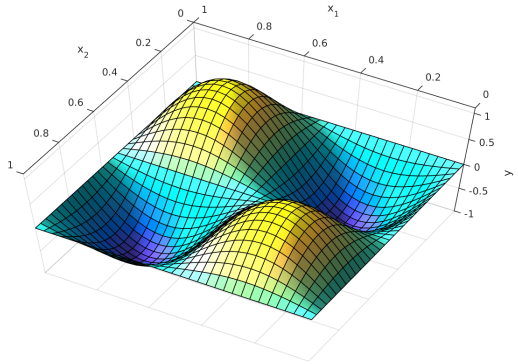
4-2 CMAC coding

Tile coding is a form of coarse coding in which the receptive fields of features are grouped into exhaustive partition of the input space. These groups are referred to as tilings, and each element is called a tile. This idea is an integral part of Cerebral Model Articulation Controller (CMAC) method (Albus, 1975). The feature values are stored in a weight vector table where each cell has its association cell that indicates binary membership degree of each possible state input vectors. The activated weights of a fully trained CMAC mapping are summed across to calculate the resulting output. Since the only algebraic operation that takes place is the addition, CMAC is a very computationally efficient method. The computational effort is determined by the number of resulting tilings: denser distributions require more calculations and are more computationally intensive. Action-value $\mathcal{Q}$-functions can be stored in this manner.

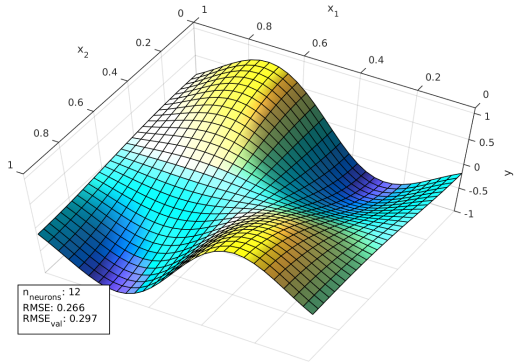
In order to encode continuous state values into tiles first define two indices i and j. Discrete index i corresponds to tile coordinate within one tiling. Given a variable x that needs to be discretised in one-dimensional space, the floor operator $\lfloor x \rfloor$ is defined as:

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\} \tag{4-19}$$

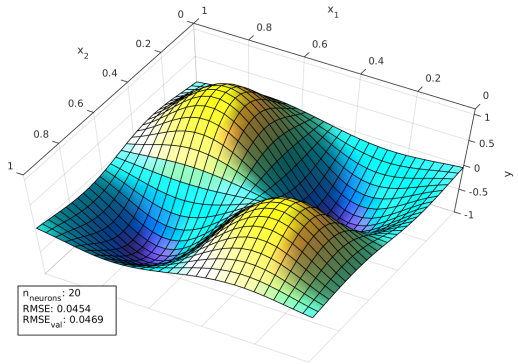
In equation (4-19) the value of $\lfloor x \rfloor$ can take negative or positive values. But the tile index i has to be compatible with positive-only indexing systems $\{i \in \mathbb{Z} \mid 0 \leq i \leq M\}$. Furthermore, given $M-1$ discretisation regions per tiling on the interval $[a, b] = \{x \mid a \leq x \leq b\}$ the variable has to be scaled using some scaling function $\phi(x)$. This scaling can be done in a variety of ways, depending on the application. The most simple way is to have a uniform, linear scaling factor as shown in (4-20). Although non-linear scaling functions can also be



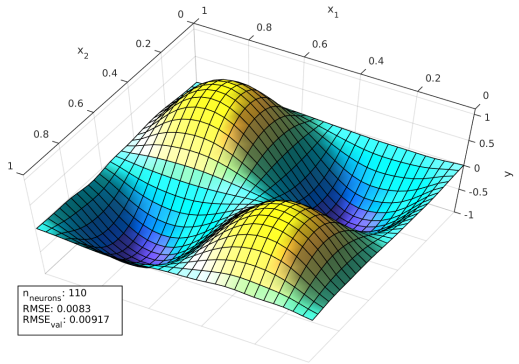
(a) Original function



(b) 12 neurons



(c) 20 neurons



(d) 110 neurons1

Figure 4-4: Trained RBF neural nets

applied, for example in order to achieve finer discretisation for certain values of x. The tile index i^p can be calculated given a continuous variable value x^p on the interval $[a, b]$:

$$\phi(x) = \frac{x-a}{b-a} \tag{4-20}$$

$$i^p = \lfloor M \times \phi(x^p) \rfloor \tag{4-21}$$

This gives the most simple way to implement discretisation in one-dimensional space. Several discretisations can then be combined in order to achieve generalization between them. Assume that there are N tilings, where j stands for tiling index $\{j \in \mathbb{Z} \mid 0 \leq j \leq N-1\}$. Then each tiling could be offset by some value Δ_j:

$$\Delta_j = \frac{1}{N} \tag{4-22}$$

$$i^p_j = \lfloor M \times \phi(x^p) + j \times \Delta_j \rfloor \tag{4-23}$$

This methodology can be extended further to facilitate encoding of multi-dimensional values of $\bar{x} = [x_1, x_2, \ldots, x_{n-1}, x_n]$, with n-dimensions. Recursion is applied in order to create a one-dimensional index $\bar{i}^p_j$ from a multi-dimensional vector value $\bar{x}^p$, as follows:

$$i^p_{j,q} = \lfloor M_q \times \phi(x^p_q) + j \times \Delta_j \rfloor \tag{4-24}$$

$$\bar{i}^p_j = \sum_{q=2}^{n} i^p_{j,q} + i^p_{j,q-1} \times M_q \tag{4-25}$$

Now that an indexing scheme had been outlined, the resulting tiling can be used to store a generalized approximation of any arbitrary function. A multidimensional array of weights w is allocated to store this generalization. Summing up the weights, located at indices derived using equation (4-25), allows to calculate the output of the approximated function:

$$y(\bar{x^p}) = \sum_{j=1}^{N} w_{\bar{i}^p_j} \tag{4-26}$$

The CMAC can be updated by comparing the output of the function $y(\bar{x}^p)$ with the desired output value y^p_t and using the resulting error to update the weights.

$$\Delta_w = y^p_t - y(\bar{x}^p) \tag{4-27}$$

$$w_{\bar{i}^p_j} \leftarrow w_{\bar{i}^p_j} + \alpha \Delta_w, \tag{4-28}$$

where α denotes the learning rate. The total required memory size for this type of array is $N \times \prod_{q=1}^{n} M_q$. This size increases linearly with the increasing number of layers, polynomially with the number of discretizations, and exponentially with the number of variable dimensions. It can grow quite large, and some trade-off has to be made about function approximation power, generalization characteristics and the required memory size. Furthermore, networks with a lot of weights can take longer to train, since ideally, each weight in the network has to be updated at some point.

CMAC training demonstration

A simple two-dimensional function was trained to demonstrate the approximation power of a CMAC net. The function used is shown in (4-29). Figure 4-6(a) illustrates the output of the function.

$$f(\bar{x}) = \sin(2\pi x_1)\cos(2\pi x_2) \tag{4-29}$$

The output of the function is shown in Figure 4-6(a). This function was approximated using several CMAC configurations. Identification and validation data sets were generating calculating the function output value for 500 randomly generated inputs.

The smallest dimension that the function can discern is inversely related to product of the number of tilings and the number of discretisations $\Delta_w \sim 1/(N \times M)$. So this product was kept constant, with $\Delta_w = 1/32$. The results of estimating the function (4-29) are shown in Figure 4-5. Trained CMAC outputs are shown in figures 4-6(b)- 4-6(d)

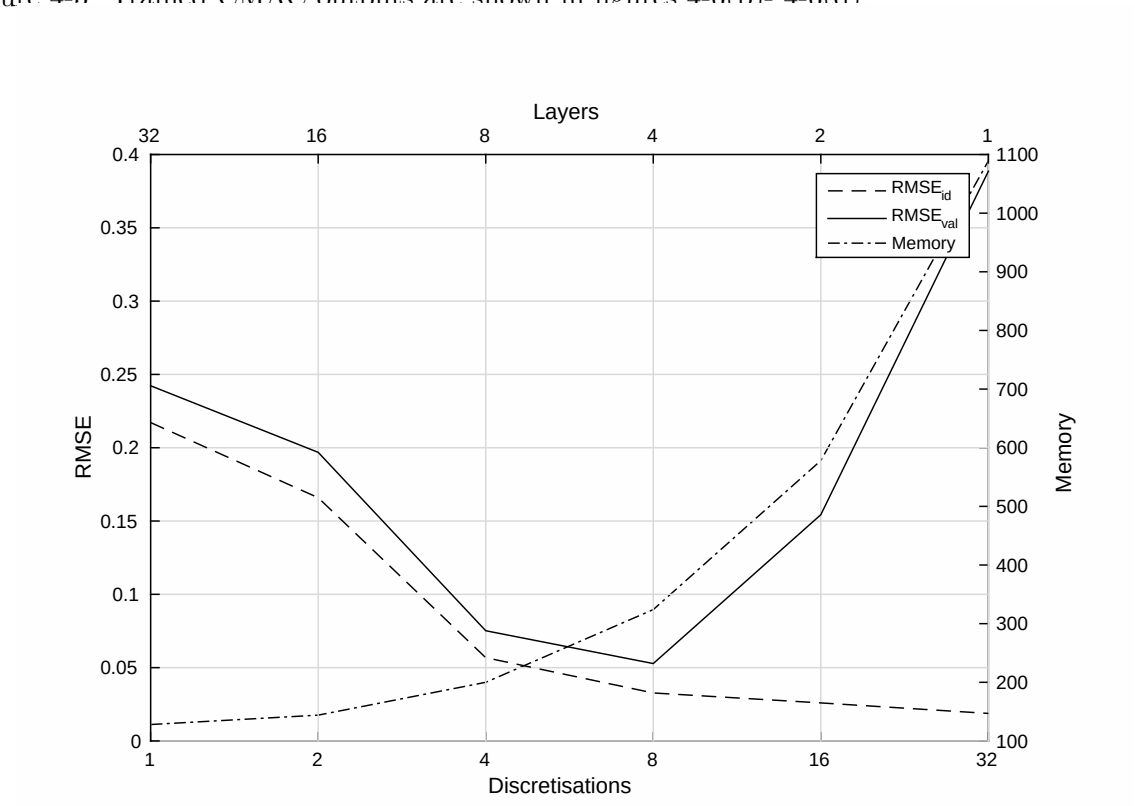
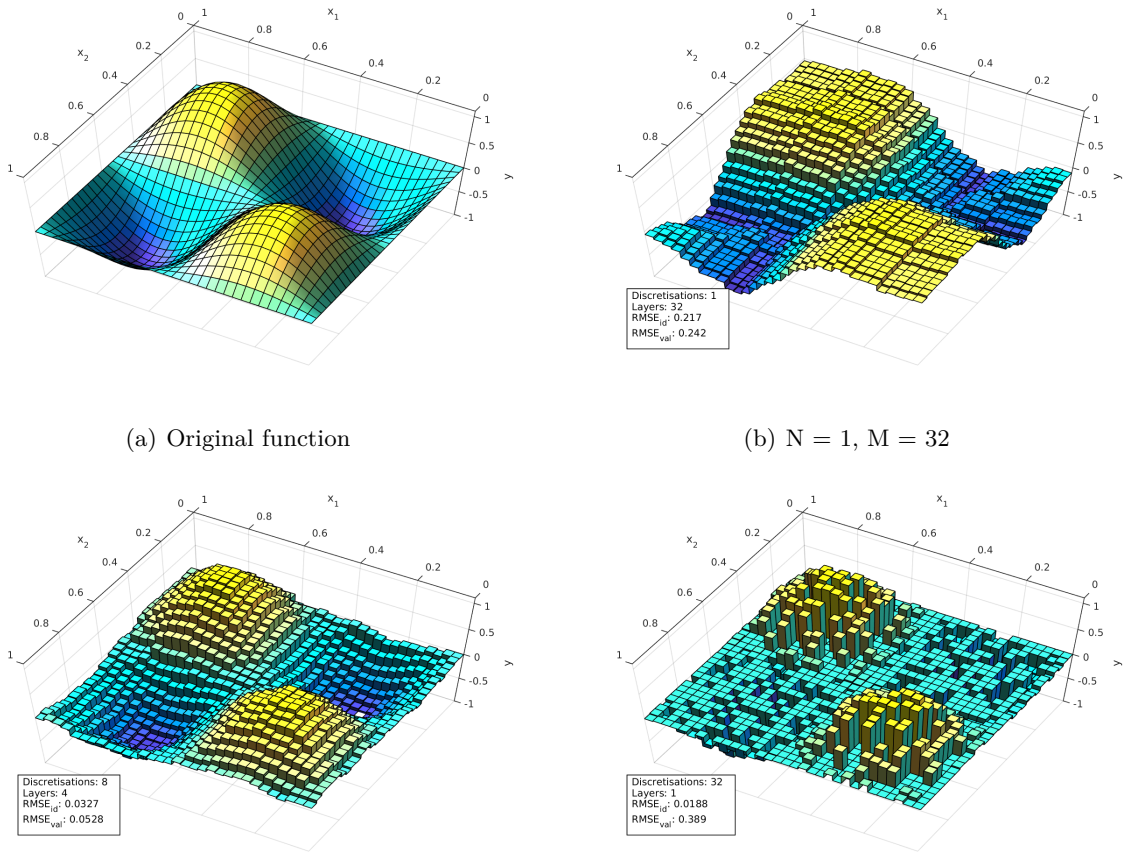


Figure 4-5: RMSE and memory size of various CMAC net configurations

Figure 4-5 shows that as the number of discretizations grows while decreasing the number of layers, the required memory size grows as well. Furthermore, at some point the validation set RMSE starts to increase with increasing number of discretizations, indicating that an optimal configuration is reached at four layers, and eight discretizations per layer. Figure 4-6(b) shows output of a CMAC with just one discretization and 32 layers. Due to insufficient resolution of the CMAC net, in this case, heavy aliasing occurs, making it impossible



(a) Original function

(b) N = 1, M = 32

(c) N = 4, M = 8

(d) N = 32, M = 1

Figure 4-6: Trained CMAC nets

to distinguish the features, present in the original function. Figure 4-6(d) shows a purely discretized approximation with only one layer. The gaps in the approximation are present due to limited generalization capacity of this configuration: insufficient identification data leads to an approximation with a very high validation error. Figure 4-6(c) shows the optimal CMAC configuration, which evidently approximates the original function rather well, but less so than an RBF neural net. Nevertheless, when making a forward pass with a CMAC net, the only arithmetic operations are addition and multiplication. Furthermore, the computational complexity depends only on the number of layers used. High fidelity can be achieved by increasing the number of discretizations at no computational cost, with the negative side-effects being that the network would require more allocated memory and might need more identification data to train.

(a) Discrete state transitions

(b) Sampled continuous state transitions

Figure 4-7: One-step state transitions for discrete and continuous state cases

4-3 Discretised generalisation and continuous state sampling

CMAC generalization offers similar performance and mechanics to continuous state function approximation methods such as Neural Networks. In practice, however, when applying it to $\mathcal{Q}$-learning problems some adjustments have to be made to it. The problem stems from the core mechanics of the $\mathcal{Q}$-learning algorithm: it relies on the difference of $\mathcal{Q}$-values of different state-action pairs, after a state transition took place. However, when a discretised generalisation is used, continuous state transition may not register unless the system actually passed the boundary between one discrete region and another. This problem will be illustrated by means of an example.

Given a one-dimensional state problem, a continuous state can be discretised with a finite number of regions. If the system state transitions are discrete in nature, a transition of state would always result in a change of its discretised approximation. However, if the state is continuous, it may take several sampled state transitions to move from one discretised region into another. Figure 4-7 shows an example, comparing these effects for a discrete-state system and a continuous-state system sampled at regular intervals, when performing 1-step $\mathcal{Q}$-learning update.

In discrete case, transitioning from state s_1 to state s_2 results in a change of discrete $\mathcal{Q}$-function approximation value. This is illustrated by figure 4-7(a). When performing update of $\mathcal{Q}$-value for state transition $s_1 \rightarrow s_2$ the following equation can be used:

$$\mathcal{Q}(s_1, a_1) \leftarrow \mathcal{Q}(s_1, a_1) + \alpha[r(s_2) + \gamma \max_a \mathcal{Q}(s_2, a) - \mathcal{Q}(s_1, a_1)] \tag{4-30}$$

As a result, after several iterations the $\mathcal{Q}$-value $\mathcal{Q}_1 = \mathcal{Q}(s_1, a_1)$ converges to:

$$\mathcal{Q}_1 \rightarrow \gamma \mathcal{Q}_2 + r_2, \tag{4-31}$$

where $\mathcal{Q}_1 = \mathcal{Q}(s_1, a_1)$, $\mathcal{Q}_2 = \max_a \mathcal{Q}(s_2, a)$ and $r_2 = r(s_2)$. When the state transition is continuous, however, several state transitions may happen within the same discrete region, as illustrated by figure 4-7(b). In that case, if equation 4-30 is applied, it actually results in two distinct outcomes for the updated $\mathcal{Q}$-value:

$$\mathcal{Q}(s_1, a_1) \leftarrow \mathcal{Q}(s_1, a_1) + \alpha[r(s_2) + \gamma \max_a \mathcal{Q}(s_2, a) - \mathcal{Q}(s_1, a)] \tag{4-32}$$

$$\mathcal{Q}(s_2, a_2) \leftarrow \mathcal{Q}(s_2, a_2) + \alpha[r(s_3) + \gamma \max_a \mathcal{Q}(s_3, a) - \mathcal{Q}(s_2, a_2)] \tag{4-33}$$

However, the values of $\mathcal{Q}(s_1, a_1)$ and $\mathcal{Q}(s_2, a_2)$ are equal in discrete domain, $\mathcal{Q}(s_1, a_1) = \mathcal{Q}(s_2, a_2) = \mathcal{Q}_1$. As a result, there are two conflicting $\mathcal{Q}$-values that the approximation may converge to:

$$\mathcal{Q}_1 \rightarrow \gamma \mathcal{Q}_1 + r_2 \tag{4-34}$$

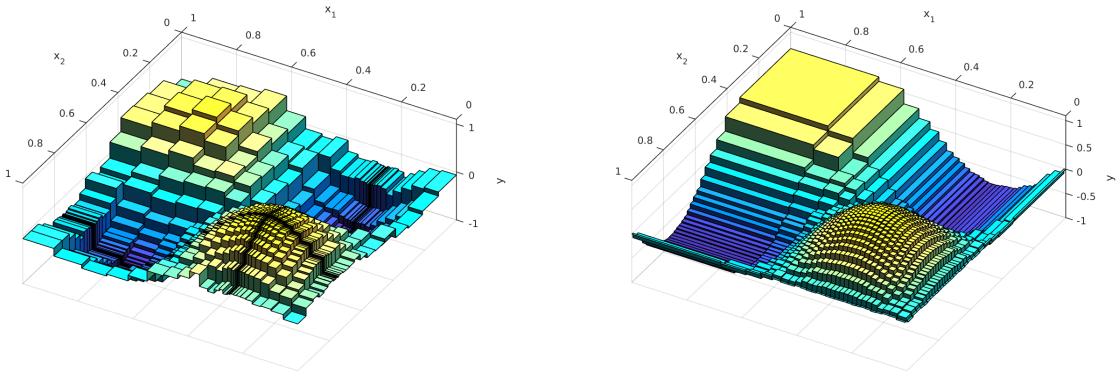
$$\mathcal{Q}_1 \rightarrow \gamma \mathcal{Q}_2 + r_3 \tag{4-35}$$

This may lead to a problem: note how the update resulting from transition between states s_1 and s_2 only takes the reward into account and ignores the $\mathcal{Q}$-value of the adjacent state. This can be shown by re-arranging equation 4-34:

$$\mathcal{Q}_1 \rightarrow \frac{r_2}{1 - \gamma} \tag{4-36}$$

This result is incompatible with $\mathcal{Q}$-learning 1-step update algorithm. As a solution, an additional condition that accounts for these effects must be added: an update may only be performed when a state transition results in transition between two discretised state approximations, otherwise the $\mathcal{Q}$-function approximation will fail to converge. In the context of using CMAC, a $\mathcal{Q}$-value function update may only be performed when two consecutive continuous state samples have distinct tile hashings, as illustrated by the following algorithm:

Data: Initialize $\mathcal{Q}(s, a)$ arbitrarily
for *each episode* **do**
  Initialize s
  for *each step* **do**
    Choose action a from s using policy derived from $\mathcal{Q}$
    while *discretised state* $S(s) == S(s')$ **do**
      take action a, observe new state s'
    end
    observe reward r,
    $\mathcal{Q}(s_k, a_k) \leftarrow \mathcal{Q}(s_k, a_k) + \alpha[r_{k+1} + \gamma \max_a \mathcal{Q}(s_{k+1}, a) - \mathcal{Q}(s_k, a_k)]$
    $s \leftarrow s'$
    if *goal is reached* **then**
      break
    end
  end
end



(a) CMAC with cubic scaling:N=8, M=4, r=3

(b) CMAC with log-sig scaling:N=8, M=4, k=2

Figure 4-8: CMAC networks trained with inputs scaling

4-4 State scaling

The network inputs can be pre-scaled before performing a forward pass. This allows specifying a focus region for the network with higher resolution, which can be beneficial to the application because it allows saving on the number of weights. Equation (4-37) shows a case of using a root function to encode an input variable. There, the input variable $x \in [a, b]$ is scaled as $x_s \in [0, 1]$

$$x_s = \begin{cases} \frac{1}{2}\left(1 - \sqrt[r]{\frac{h-x}{h-a}}\right) & \text{if } x < h \\ \frac{1}{2}\left(1 + \sqrt[r]{\frac{x-h}{b-h}}\right) & \text{if } x >= h \end{cases} \tag{4-37}$$

The variable x_s denotes a scaled version of a variable x. The offset factor h denotes the "focus" of the scaling, e.g. the values for which the scaling results in a grid that is finer. This sets the region around which the scaling gradient is the steepest. The variable r denotes the scaling power. When it's set to one, the scaling is purely linear. The same function used previously was trained using cubic scaling, with $r = 3$ to demonstrate the effect. The resulting fully trained CMAC output is shown on Figure 4-8(a). The focused area was set at $x_f = [0.25, 0.75]$.

Another way to scale the variable x is to apply a sigmoid function, such as the log-sigmoid, shown in equation (4-38). Sigmoid scaling allows more flexibility than only using a scaling power factor. In particular, it allows to set up the tiling "gradient" in the focused region. This gradient is set using the tangent-sigmoid function derivative, shown in equation (4-39).

$$g(x_s) = \frac{1}{1 + \mathrm{e}^{-4k(x_s - h)}} \tag{4-38}$$

$$g'(x_s) = \frac{4\,k\,\mathrm{e}^{4\,k\,(h-x)}}{\left(\mathrm{e}^{4\,k\,(h-x)} + 1\right)^2} \tag{4-39}$$

$$g'(h) = k \tag{4-40}$$

Equation (4-40) shows the maximum gradient of the log-sigmoid function, achieved when $x_s = h$. This gradient can be set manually, depending on how much "zoom" is desirable at the region of interest. The output of a function, trained with $x_f = [0.25, 0.75]$ and $k = 2.5$ is shown in Figure 4-8(b).

This example shows that application of scaling allows achieving a much finer discretization at the specified region of interest while generalizing the states further from the focus area.

4-5 Encoded RBF-based neural net

The CMAC has an advantage over a "pure" implementation of radial basis function neural network, where all weights are activated with each forward pass and all weights are updated at each backward pass. In the CMAC only the local weights, situated close to the state, are activated and updated, and finding it's output is as simple as summing the weights without any need for using an activation function. Nevertheless, the nature of the activation function used in an RBF net makes it possible to optimize it for computational performance, even surpassing CMAC in some ways.

A typical Gaussian radial basis function is based on the principle that its output value diminishes as its input moves further away from the neuron center. This property also means that at a certain distance from the neuron center the neuron output contribution is negligible, compared to the neurons activated near the input. Therefore, the neurons situated further away from the activated region can be safely neglected, since they have little influence on the output of the neural net in that region. By limiting the number of activated neurons the size and resolution of the network can be decoupled from real-time performance. If only a limited number of neurons is activated, it does not matter anymore how many neurons there are in total. This situation still poses a problem, however, since it must be necessary to be able to determine which neurons are in fact the closest to the activated region. When the neurons are irregularly spread throughout the state-space region, this might be difficult, since the distance to each neuron must be calculated to determine whether it has to be activated or not. On the other hand, if the neurons are evenly spaced throughout the state range, selecting the neurons that have to be activated becomes very easy. The activated neurons are known to be situated on a regular grid, therefore to determine which ones must be activated a simple encoding scheme can be applied, similar to the one described in the previous section dealing with the CMAC. This principle is illustrated in figure 4-9.

On figure 4-9 only the neurons situated within some maximum radius $r_{lattice}$ are activated. The distribution of the activated neurons always follows the same type of circular lattice pattern, offset from the coordinate of the current input state. This pattern can be calculated befre network initialization, stored and reused. These patterns are stored as an offset from the neuron, situated the closest to the input state, denoted as $\bar{c}_{i,j,\ldots,n}$, where n denotes the network input dimensionality. The network input can be converted to a positive-only index, using methodology described in section 4-2. One difference, however, is that whereas for the CMAC encoding a floor operator $\lfloor x \rfloor$ is used, in neural encoding a rounding operator $rpi(x) = \lfloor x + \frac{1}{2} \rfloor$ is used instead to find the coordinates of the closest neuron rather than tile coordinate. The following encoding principle can be illustrated the easiest for a single-input coded neural network with activation radius $r_{lattice}$. The lattice offset coordinates for such a

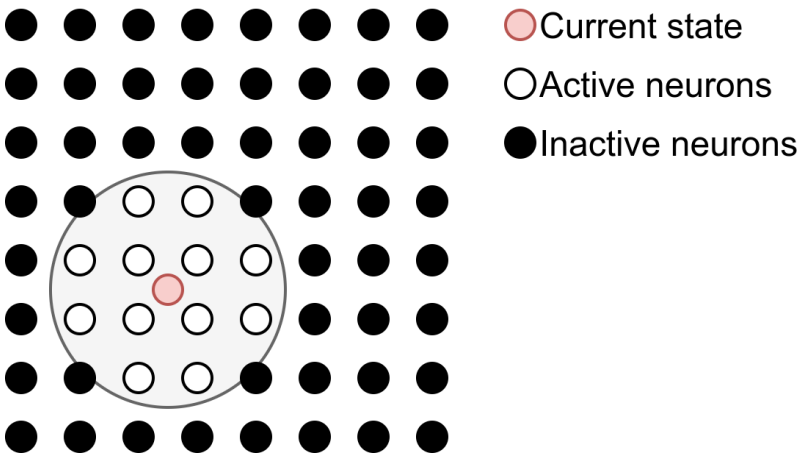


Figure 4-9: Neuron activation scheme

network can be represented as

$$c_{lattice} = [-R, \ldots, 0, \ldots R] \tag{4-41}$$

$$R = \lfloor r \rfloor \tag{4-42}$$

Given an input x, the coordinate of the nearest neuron $\bar{c}$ is found:

$$\bar{c} = rpi(x) \tag{4-43}$$

Then the activated neuron coordinates and the neuron input offsets $\Delta_i = x - \mathbf{c_i}$ become:

$$c_{neur} = \bar{c} + [-R, \ldots, 0, \ldots, R] \tag{4-44}$$

$$\Delta = (x - \bar{c}) + [-R, \ldots, 0, \ldots, R] \tag{4-45}$$

The entire procedure is quite simple and allows to calculate the neuron coordinates as well as inputs for each neuron simultaneously and efficiently. This methodology could also be easily extended to higher input dimensions by generating multi-dimensional neuron offset lattices. With each additional dimension, the number of required neurons grows. Table 4-1 shows the number of activated neurons for a value of $r = 2.4$ and the percentage of the total neurons in the network for an encoding that consists of 8 neurons per-dimension.

		N_a activated neurons (% total)			
N_{inputs}	$N_{neurons}$	$r = 1.0$	$r = 1.5$	$r = 2.0$	$r = 2.4$
1	8	3 (37.5 %)	3 (37.5 %)	5 (62.5 %)	5 (62.5 %)
2	64	5 (7.81 %)	9 (14.0 %)	13 (20.3 %)	21 (32.8 %)
3	512	7 (1.37 %)	19 (3.71 %)	33 (6.44 %)	57 (11.13 %)
4	4096	9 (0.22 %)	33 (0.81 %)	89 (2.17 %)	137 (3.34 %)

Table 4-1: Number of activated neurons for selected dimensions

The number of activated neurons grows with increased radius and input dimensionality. For a single-input neural net, the effect of neuron encoding is quite small, and the extra computational load might offset any benefits. For a larger number of inputs, however, there is a drastic increase in efficiency. For a neural net with three inputs, encoded with eight neurons per-dimension and activation radius $r = 2.4$ only 11.13 % of all neurons have to be activated at any forward or backward pass. Increasing the resolution or the number of dimensions would result in an even more drastic increase of efficiency.

4-6 Discussion

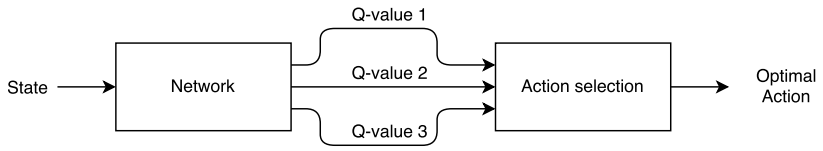
This chapter described a methodology that could be used to apply $\mathcal{Q}$-learning to problems with continuous states and actions, using function approximation to store the value function. Two algorithms were described: neural networks and CMAC tile coding. Both were demonstrated to be capable of the task, and their distinct advantages and disadvantages were discussed. While RBF neural nets show superior approximation power, CMAC is less demanding computationally. An approach that combines the efficiency of state encoding with the approximation power of a neural network was demonstrated as well. It was shown that defining activation regions for neurons inside of a neural network can result in drastic improvement of efficiency, especially for multidimensional input states and networks defined by a large number of neurons within the hidden layer.

Chapter 5

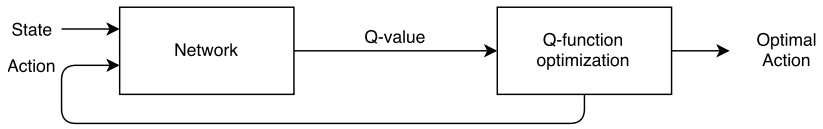
Optimization methods

The underlying principle of Q-learning requires a method that is capable of finding an action that results in the optimal value of the Q-function. Various methodologies can be applied, depending on the type of generalization function used to store the Q-function approximation. This chapter outlines some selected approaches to this problem.

Given a continuous-state Q-learning problem, there are two methods to find the optimal action: using a discrete set or a continuous range of actions. A schematic, shown in figure 5-1 illustrates these two approaches.



(a) Discrete action



(b) Continuous action

Figure 5-1: $\mathcal{Q}$-learning action selection approaches

In a discrete-action approach, the network, used to generalize the Q-function, produces several Q-value outputs for any given state, each of them associated with a predetermined action. Once these values become available, the action that results in the highest output is selected as the optimal one. In a continuous approach, both the state and the action are used as inputs for the network. The output value is then optimized to find the optimal action. The discrete approach has an advantage over continuous approach because when using it only one forward pass of the network is required, whereas the continuous method typically requires several passes. This means less computational load when implemented, and it also makes

the whole method easier to implement. On the other hand, a continuous approach allows to select a more precise action, and typically the actions themselves transition more smoothly. This offers an advantage when controlling a sensitive system such as a Quadrotor.

To calculate the action that results in an optimal Q-value when using continuous actions, some function optimization technique has to be applied. Optimization techniques play an important part in the engineering process. Many real-world systems are highly uncertain and non-linear, so finding an optimal solution to a given problem using purely analytical means is not always possible. Therefore, a variety of tools is applied to solve these types of problems: linear programming, integer programming and heuristic methods, among many others. Meta-heuristics is a recently developed family of optimization methods. These methods provide a general structure and strategy guidelines to find a general solution for a particular kind of problem (Hillier, Lieberman, & Newton, 2015). One of such methods is the Particle Swarm Optimisation (PSO) algorithm, described in this chapter. It applies to many types of systems, and what makes it interesting is that it is capable of dealing with most functions, even if they are not continuous or non-differentiable, such as CMAC networks. Another method discussed is optimization based on local function approximation of the Q-value.

5-1 Particle Swarm Optimisation

The Particle Swarm Optimisation (PSO) is a stochastic, population-based optimization technique, in which several candidate solutions coexist and collaborate. PSO draws its inspiration from biological systems. The particle swarm concept is based on flocks of birds shoals of fish, colonies of bees and ants (Poli, 2007).

Due to its simplicity and ease of implementation and adaptation, PSO is widely employed in the areas of engineering, statistics, management, biology, etc (Poli, 2008). PSO has some advantages over similar meta-heuristic techniques, such as Genetic Algorithm(GA). It is well suited for problems that involve continuous states, although discrete variations of it also exist. It is relatively inexpensive to implement, computationally (Mukherjee & Ghoshal, 2007), and thus suitable for online applications (Iruthayarajan & Baskar, 2009). Overall, PSO performs well when applied to unconstrained nonlinear optimization problems with continuous design variables (Hassan & Cohanim, 2005).

The main principle behind it is the simulation of the social behavior. Each solution is called a "particle", and it is represented as a member of a multi-dimensional vector $a_j = (a_{j,1}, a_{j,2}, \ldots, a_{j,g})$ in the g-dimensional space. Each particle has several components: it's location within the solution search space, rate of position change (velocity) and inertia weight. In addition, each particle has a "memory". The particle coordinate corresponding to the best solution previously encountered by the particle is preserved in another multi-dimensional vector $a_j^* = (a_{j,1}^*, a_{j,2}^*, \ldots, a_{j,g}^*)$. Furthermore, the particle coordinate that corresponds to the best solution previously encountered by any particle within a swarm is represented by $a^{sw} = (a_1^{sw}, a_2^{sw}, \ldots, a_g^{sw})$.The velocity of each particle is adjusted at every update step, depending on the position of the most fit particle in the group, as well as the location of the best previously encountered solution for each particle (Gaing, 2004). Each particle travels through the pre-determined search space with some velocity v_j, which is represented as a vector in g-dimensional space: $v_j = (v_{j,1}, v_{j,2}, \ldots, v_{j,g})$. The initial values for particles a_j

and velocities v_j can be initialised randomly, within some feasible range. Velocity v_j of each particle is updated at each algorithm iteration, based on the coordinate of the best particle of the swarm a^{sw}, and the coordinate of the best solution previously encountered by the current particle a_j^*. The particles coordinates a_j are then updated, based on new particle velocity v_j:

$$\begin{aligned} v_{j,g}^{k+1} &\leftarrow w \cdot v_{j,g}^k + c_1 r_1 (a_{j,g}^* - a_{j,g}^k) + c_2 r_2 (a_g^{sw} - a_{j,g}^k) && \text{(5-1)} \\ a_{j,g}^{k+1} &\leftarrow a_{j,g}^k + v_{j,g}^{k+1} && \text{(5-2)} \\ j &= 1, 2, \ldots, n \\ g &= 1, 2, \ldots, m \end{aligned}$$

Where

n	number of particles in the swarm
m	number of particle dimensions
k	current iteration number
p_j^k	coordinates of jth particle in the search space at iteration k
p_j^*	best solution previously encountered by the particle
p^{sw}	best solution from the entire swarm
v_j^k	velocity of jth particle at iteration k
w	inertia weighting factor $w = 1 - \frac{k}{k_{max}}$
c_1, c_w	acceleration constants set in the range $[0, 1]$
r_1, r_2	random numbers in the range $[0, 1]$

Collectively the particles move towards the better solution, while also covering the search domain. The components c_1 and c_2 in equation (5-1) weight the stochastic acceleration terms towards either $a_{j,g}^*$ or a_g^{sw}. These determine how far the particles are allowed to venture in search of a solution, before losing their momentum and returning to the initial location. Lower values allow the particles to roam further. The inertia factor w allows to control the impact of the previous trajectory of the particle, influencing the exploration pattern, Larger inertia facilitates a more global search scope, while less inertia serves to fine-tune the locally optimum solution. The factor is usually varied throughout the exploration process. The velocity of each particle is generally bound within some limit $V_g^{min} \leq v_{j,g}^k \leq V_g^{max}$. If this limit is too high, the particle might miss the good solutions, if it is too small, it might not explore sufficiently beyond its current location.

After each particle velocity and coordinate update, new solution values evaluated using some objective function $g(a)$ that needs to be optimised. The value of $J(a_j)$ corresponds to the output of the optimization function, with the coordinates of particle a_j as the input. Then the values of $J^* = (J(a_1^*), J(a_2^*), \ldots, J(a_n^*))$ and $J^{sw} = \max(J^*)$ can be updated, using new particle coordinates a:

$$\begin{aligned} J_j^* &\leftarrow \max(J_j^*, J(a_j^{k+1})) && \text{(5-3)} \\ J^{sw} &\leftarrow \max(J_j^*, J^{sw}) && \text{(5-4)} \end{aligned}$$

If the current particle output $J(a_j^{k+1})$ is higher than the best previously encountered output J_j^* then this value is updated, and the particle coordinates are preserved as $a_j^* \leftarrow a_j^{k+1}$.

Likewise, if output $J(a_j^{k+1})$ is higher than that of the entire swarm J^{sw} then the value of J^{sw} is updated and coordinates of the particle are preserved $a^{sw} \leftarrow a_j^{k+1}$.

PSO can be applied to multi-dimensional problems, and it can also be implemented with non-differentiable and nonlinear functions, such as the CMAC optimization. It can be incorporated into value function-based reinforcement learning process, as illustrated by the following algorithm:

Data: Initialize a_i, v_i arbitrarily
Define $a^* = \arg\max_a \mathcal{Q}(s, a)$
Define $V(s) = \mathcal{Q}(s, a^*)$
while *current iteration* $<$ *iteration limit* **do**
  for *each particle* a_j **do**
    Update particle velocity v_j^{k+1} using (5-1)
    Update particle coordinate a_j^{k+1} using (5-2)
    Evaluate the $\mathcal{Q}$-function $\mathcal{Q}(s, a_j)$, where particle a_j represents the selected action a
    if $\mathcal{Q}(s, a_j) > \mathcal{Q}_j^*$ **then**
      Update current particle optimum value and particle coordinate
      $\mathcal{Q}_j^* \leftarrow \mathcal{Q}(s, a_j)$
      $a_j^* \leftarrow a_j$
    end
    if $\mathcal{Q}(s, a_j) > \mathcal{Q}^{sw}$ **then**
      Update swarm optimum value and particle coordinate
      $\mathcal{Q}^{sw} \leftarrow \mathcal{Q}(s, a_j)$
      $a^{sw} \leftarrow a_j$
    end
    if *solution converged* **then**
      break
    end
  end
end
return $a^* = a_{sw}$, $V(s) = \mathcal{Q}_{sw}$

This algorithm allows calculating the optimal action a^* at any given state s, as well as value function output $V(s)$. It can work with any number of action states and should converge to a global optimum, given enough particles.

5-2 Value function optimization using root-finding

The use of exponential-based Radial Basis Function in neural networks offers a significant advantage over the use of CMAC: the RBF activation function is twice-differentiable, and as such there are some efficient optimization methods available to maximize its value.

5-2-1 Polynomial fit

One of the methods to find the maximum of any function is to use polynomial curve fitting. Consider a function in the form

$$Y = A(x) \cdot \theta, \tag{5-5}$$

where $A(x)$ is a matrix where each row represents a polynomial in the form

$$a(x) = \begin{bmatrix} 1 & x & \dots & x^N \end{bmatrix} \tag{5-6}$$

for an Nth-degree polynomial, and θ contains the polynomial coefficients

$$\theta = \begin{bmatrix} p_0 & p_1 & \dots & p_N \end{bmatrix}^T \tag{5-7}$$

The first derivative of this polynomial can be described as

$$\dot{Y} = \dot{A} \cdot \theta \tag{5-8}$$

where each row of $\dot{A}(x)$ is defined as

$$\dot{a}(x) = \begin{bmatrix} 0 & 1 & \dots & Nx^{N-1} \end{bmatrix} \tag{5-9}$$

The maxima/minima of the function, described in 5-5 are reached where the values of it's derivative $\dot{a}(x)$ reach zero, e.g. at it's roots r. This can be done by calculating the roots of function $\dot{Y}$, by finding eigenvalues of the companion matrix $\dot{A}^*$

$$\dot{A}^* = \begin{bmatrix} -2\frac{p_2}{p_1} & -3\frac{p_3}{p_1} & \dots & -N\frac{p_N}{p_1} \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \tag{5-10}$$

$$r = eig(\dot{A}^*) \tag{5-11}$$

Each of the resulting polynomial roots in r could indicate either the maximum or the minimum of the function. So the function 5-5 must be re-evaluated at each root found in r to determine the maximum value.

This methodology could be extended beyond finding extremes of a simple polynomial. Using least-square approximation a polynomial function approximation can be fitted to any set of data or a function. Consider a function $f(x)$. In order to approximate it with a polynomial of degree N it must be sampled at values $\bar{x} = \begin{bmatrix} x_0, x_1, \dots, x_M \end{bmatrix}$, where each sample represents function input x_m and the outputs of the function are denoted by $\bar{Y} = [f(x_0), f(x_1), \dots, f(x_M)]$. There must be at least as many samples as there are degrees of freedom in θ, $N \leq M$. With that, the parameter vector $\bar{\theta}$ can be calculated by solving the system using linear regression

$$\bar{\theta} = (A(\bar{x})^T A(\bar{x}))^{-1} A^T \bar{Y} \tag{5-12}$$

With the parameters of the approximation polynomial determined, the procedure outlined previously can be used to approximate the local maxima/minima of the function $f(x)$. The accuracy of such an approximation relies heavily on the quality of fit of the polynomial. For a function that behaves more "polynomial-like," this results in a highly accurate estimate, whereas for functions that behave differently this may lead to errors.

5-2-2 RBF fit

A function optimization methodology, similar to the one outlined previously, can in principle be used with any sort of approximation function besides a polynomial, for as long as the function is differentiable. In fact, it is beneficial to use an approximator that mimics the approximated function more closely, since it gives more accurate results due to a better fit. Using polynomial curve fitting has its advantages: it's a computationally cheap way to compute the maximum, and it can be solved analytically. Whereas not every fitted approximation function would have a purely analytical solution. Nevertheless, there are other methodologies to deal with it, such as Newton's iteration method.

Consider a function in the form

$$Y = A(x)\theta \tag{5-13}$$

where each row in A represents a set of RBF neuron outputs

$$a(x) = \begin{bmatrix}\phi_0(x) & \phi_1(x) & \dots & \phi_N(x)\end{bmatrix}, \tag{5-14}$$

$$\phi_i(x) = \exp(-(x - \mathbf{c}_i)^2), \tag{5-15}$$

and θ is the parameter vector that contains network output weights $\theta = [w_0, w_1, \dots, w_N]$. The activation function in 5-15 can be differentiated twice with respect to input x:

$$\dot{\phi}_i(x) = -2(x - \mathbf{c}_i)\phi_i(x) \tag{5-16}$$

$$\ddot{\phi}_i(x) = -2(1 - 2(x - \mathbf{c}_i)^2)\phi_i(x) \tag{5-17}$$

This allows to find the derivatives of the functions in 5-16 and 5-17:

$$\dot{a}(x) = -2\begin{bmatrix}(x - \mathbf{c}_0) & (x - \mathbf{c}_1) & \dots & (x - \mathbf{c}_N)\end{bmatrix} \times a(x) \tag{5-18}$$

$$\ddot{a}(x) = -2\begin{bmatrix}1 - 2(x - \mathbf{c}_0)^2 & 1 - 2(x - \mathbf{c}_1)^2 & \dots & 1 - 2(x - \mathbf{c}_N)^2\end{bmatrix} \times a(x) \tag{5-19}$$

Note that the exponential component in 5-15 and the offsets $(x - \mathbf{c})$ can be reused when finding the derivatives of the activation function, which means that the computational load can be reduced when implementing these calculations in software.

Now the principle of finding the minima and the maxima of a function, expressed as an RBF net, remains the same: the points where it's derivative reaches zero must be identified. However, an RBF net does not have an analytical solution such as a polynomial function. Here, Newton's iteration method could be used instead. Newton's iteration is a numerical method designed to find roots(zeros) of differentiable equations in the form

$$f(x) = 0 \tag{5-20}$$

Provided an initial guess x_0, the successive update formula for this method is

$$x_{k+1} = x_k - \frac{\dot{f}(x_k)}{f(x_k)}, k = 0, 1, \ldots \tag{5-21}$$

Typically with each consecutive update the output of the function $f(x_k)$ converges closer to it's nearest zero value. This means, however, that it is only capable of finding the nearest root, one at a time. Therefore, for a function with several possible zeros several initial point must be required. When applied to the problem of finding local maxima/minima the Newton's iteration procedure can be applied as follows:

First, several samples of the optimized function $\bar{x} = \left[x_0, x_1, \ldots, x_M\right]$ must be generated throughout the search region. Outputs of the function are recorded in $\bar{Y} = [f(x_0), f(x_1), \ldots, f(x_M)]$. The parameter vector $\bar{\theta}$ can be calculated using 5-12. Then the first and the second derivatives of the approximated function $\bar{f}(x) = a(x)\bar{\theta}$ are:

$$\dot{f}(x) = \dot{a}(x) \cdot \bar{\theta} \tag{5-22}$$
$$\ddot{f}(x) = \ddot{a}(x) \cdot \bar{\theta} \tag{5-23}$$

Newton's update can be modified to find the zeros of $\dot{f}(x)$:

$$x_{k+1} = x_k - \frac{\ddot{f}(x_k)}{\dot{f}(x_k)}, k = 0, 1, \ldots \tag{5-24}$$

After evaluating several starting points the value of x that results in maximum/minimum output can be found. This approach is more computationally intensive than the previously described polynomial curve approach because several function evaluations must be performed. Nevertheless, it is more suited for optimization of neural network outputs, since an approximated model described using the same type of activation functions as the neural network can approach the function being optimized much more closely.

5-3 Discussion

This chapter had outlined some optimization methods that can be applied to continuous-state, continuous-action Q-learning. A meta-heuristic particle Swarm Optimization Method was described. PSO is a highly effective method that can be applied to a wide array of functions, but its implementation requires a large number of function evaluations, which leads to difficulty in applying it to a real-time system. Two optimization techniques based on function approximation were described as well: using polynomial function approximator and an RBF neural net. The polynomial approach is highly efficient, computationally. Another advantage of using a polynomial is that all global maxima/minima of the approximated function can be found simultaneously. On the other hand, this methodology is only effective when the optimized function behaves like a polynomial function. In the case of neural network output optimization, it might be beneficial to use RBF-based function approximation instead, since it approximates the network more closely.

Chapter 6

UAV dynamics and control scheme

The Quadrotor concept had been known since the early days of aeronautics. One of the earliest examples of such aircraft is the Breguet-Richet Quadrotor helicopter Gyroplane No.1, built in 1907. A quadrotor can be described as a vehicle that has four propeller rotors, arranged in pairs. The two pairs (1,3) and (2,4) turn in opposing directions, and by varying the thrust of each individual rotor the vehicle pitch, roll, yaw and altitude can be changed. This chapter introduces some modeling methods that can be used to simulate a quadrotor vehicle, and introduces a general mathematical model of quadrotor dynamics.

6-1 Model identification

To build a viable model of the process, the parameters of the system under control have to be identified. System identification process is aimed at describing properties of a given system and how it responds to inputs. Typically as more information about system and it's behavior is gathered, the observed regularities can be formulated into a mathematical model. This model can then be used for a wide variety of tasks: predicting system behavior and input responses under certain conditions and gaining general insight into it's physical characteristics for the purposes of design and control. The general procedure for designing a model according to Scientist rule is described as follows (Bohlin, 1994):

Data: Initialize the root model structure(smallest conceivable set of components) and set initial parameter values.
while *falsified* **do**
  Specify tentative free parameter set and fit free parameters
  while *unfalsified* **do**
    Test the model against a set of alternative structures
    if *There are no more free parameters* **then**
      Expand the set of model components
    end
    Specify alternative free parameter set
    Select parameter-free statistic
    Evaluate test statistic
    Appraise model
    if *A better model is found* **then**
      Return falsified
    end
  end
end

Creating a model is not a trivial task and it often takes a lot of time and effort. One of the goals of this research is to design a procedure that can be used to do this autonomously, i.e. an algorithm that allows to learn it by learning more about environment and it's interaction with the agent. In general, models themselves are divided in three categories:

- White-box models: there is an established physical relationship between model states, the model is constructed entirely from prior knowledge and theory
- Grey-box model: Some of the system dynamics are known, while others aren't. The overall model is augmented with parameters estimated from data to increase accuracy, compensating for missing parameters. Typically purely white box models are used for relatively simple systems, while as complexity increases grey-box elements are introduced. Since the augmentation part is built from observed data, a certain amount of testing is required to build an accurate estimation.
- Black-box model: Little is known about the model. It is built entirely using observed behavior data by analyzing it's dynamics in response to the inputs. Typically a lot of data describing the system behavior is required to build an accurate estimate.

The advantage of white box approach is that it allows for a relatively easy solution to a problem. When applied to a simple dynamic system it provides a fairly straight-forward solution, and it does not require a lot of computational power. Among disadvantages of such an approach is that as the system dynamics become increasingly complex and non-linear, it becomes more and more difficult to account for all of them. Furthermore, some of these dynamics could be impossible to account for purely theoretically due to the lack of knowledge.

Grey-box approach offers a more flexible solution. While maintaining robustness and stability of a white-box approach, it allows to capture the dynamics that are more difficult to model analytically. As the knowledge of the system increases and the "gray" area of the model

becomes more explored, the “white” component could be further developed and expanded, improving the model. Advantage of a gray-box approach is that an accurate model could be constructed with a relatively small amount of information about it. Among the disadvantages is the fact that employing grey-box approach is more computationally intensive than a purely white box one.

Black box approach allows to model highly complex systems with a high degree of accuracy. It mostly neglects the actual physical dynamics of the system, basing the model purely on test and simulation data. It is fit to describe stochastic nonlinear dynamics of almost any system. The constraints come from the fact that a large set of training data is required to build a model. Nevertheless, purely black box approach could be very computationally intensive. Another factor to account for is that when a nonlinear process is concerned, the range of possible models could be exceedingly large (Sjöberg et al., 1995) which adds extra difficulty.

In the context of the problem at hand, gray box approach is the most suitable one. In principle, the system is intended to be used to control a physical vehicle. This already gives some background to what the model should ”look” like. A typical vehicle like a multicopter UAV or an AGV can have up to 6 degrees of freedom in it’s movement. Typical sensor data that can be used to collect information about states of the vehicle consists of rotations and translations along its x,y and z axes. In addition to that, parameters like vehicle mass and dimensions could be used to estimate the kinematic dynamics of its movement. In principle this already should be enough to construct a linear time-invariant white box model.

Nevertheless, in real world the behavior of such a plant would most certainly include some non-linearities. These effects can be mitigated by more thorough examination of physical phenomena behind model dynamics, but this process is often time consuming, and it can not be generalized as each given vehicle configuration would exhibit some unique behavior. A purely black box approach is not suitable either. Slow convergence of this method means that more testing time would be required to build a model. Since the resulting model transparency is limited, it also introduces a higher degree of uncertainty about it’s performance, which makes it less suitable for the purposes of forecasting and control. Black box model is not enough for all control purposes (Bohlin, 1991). A black box component can be used to augment the white box model to account for nonlinearities, hidden states and so on, making it a gray box approach.

6-2 Modelling methods

As discussed previously, gray box type of model is most suitable for the task at hand. It can be constructed using a combination of approaches, described in this section.

Linear time-invariant (LTI) models are perhaps the most important and popular class of models used in practice today. The theoretical body of knowledge behind them is very extensive, they are computationally inexpensive and they can be applied to a wide variety of tasks. LTI models describe the behavior of linear and time-invariant systems, but they can also be applied to nonlinear systems when linearized around some condition. The linearity characteristic of such models implies that the output response to a linear combination of inputs is the same as response to each individual input. This requires some simplification in most cases. Nevertheless, in many cases LTI models can produce very good results.

LTI models can be extended by identifying linearized models around relatively close operating points and then combining them into a linear parameter varying (LPV) model, resulting in a gray-box approach. This method had been applied to control of Unmanned Water Craft(UWC) in (Svendsen, Holck, Galeazzi, & Blanke, 2012). UWCs generally exhibit very different maneuvering characteristics, depending on the running attitude of the vehicle. Applying adaptive controller based on a gray-box approach was shown to guarantee both robustness and stability of the closed loop in different operational conditions.

Neural Networks can be used to model and control very complex systems with sufficient accuracy without complete knowledge of its inner composition. Neural networks themselves can be divided into two types: multilayer neural networks and recurrent networks. Despite some differences, they could be viewed in a unified fashion as part of a broader discipline (Narendra & Parthasarathy, 1990).

When a system is very complex, creation of precise mathematical models can become quite difficult. Principle of incompatibility asserts that complexity of a system is inversely proportional to the precision with which it can be analyzed. To deal with these types of issues, *fuzzy models* can be applied. They are based on imprecise descriptions of the relationships between signals within the system, described in fuzzy, linguistic variables (L. Zadeh, 1975). The premise of this approach is that the key elements in model or logic design are not represented by pure numbers but using fuzzy sets: classes of objects for which the transition from membership to non-membership is gradual, rather than abrupt (L. a. Zadeh, 1973). This kind of approach allows to generalize the problems and to approach highly complex or ill-defined phenomena in a systematic manner. Fuzzy logic controllers are often applied to UAV control. (Kurnaz, Cetin, & Kaynak, 2010) describe fuzzy logic controller applied to bank control of a fixed-wing uav. The objective of the controller is to reach and hold the bank angle required to reach a given heading. It was demonstrated that this controller demonstrates superior performance. Nevertheless, some issues were also identified regarding stability of the learning algorithm. For some flight conditions instability of the flight algorithm may result in instability of flight performance.

6-3 Quad-copter modelling and control

Control of a typical aerial quadrotor vehicle can be split into four principal parts: pitch and roll controls, collective thrust and yaw control. Quadrotors have six degrees of freedom: three translational and three rotational, and they can be controlled along each of them. Rotational and translational motion are coupled. Tilting along any axis results in vehicle movement in the direction perpendicular to the tilt angle. Extra couplings exist between individual rotors and the aircraft body. For example to rotate along the vertical axis while maintaining altitude the distribution of thrust has to be changed in such manner that some of the rotors spinning in one direction are rotating more slowly, while the rotors spinning in the direction of intended rotation are moving faster. This results in a challenging problem: in practice it is very difficult to control a quadrotor purely manualy. In addition to difficulty of control, there is also a question of stability of multicopter craft. Multicopters are not inherently stable and damping has to be achieved actively. Electronics are used to stabilize it and distribute the thrust across the rotors.

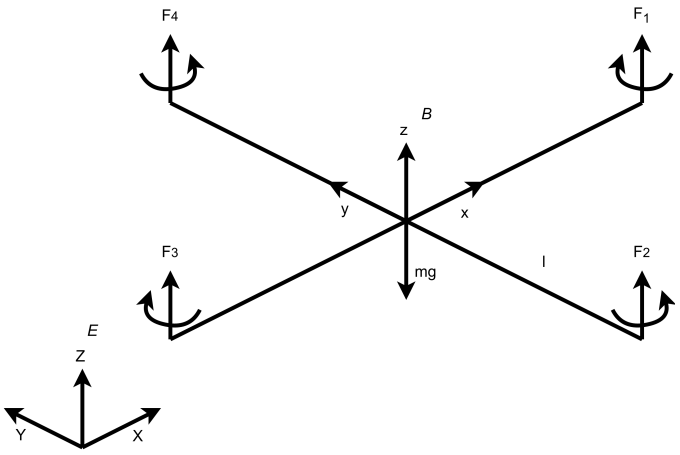


Figure 6-1: Quadrotorotor configuration schematic, body fixed frame B and inertial frame E.

6-4 Dynamic model of a quadrotor

This section will describe modelling of a quad-copter using a purely white-box approach. Schematic of a quadrotor body and inertial frames is shown in Figure 6-1.

The craft dynamics can be modeled using Lagrangian method (Bouabdallah, Noth, Siegwart, & Siegwan, 2004; Mohammadi, Shahri, & Boroujeni, 2012). The following assumptions are made when deriving the model:

- The quadrotor body structure is rigid
- The quadrotor body structure is symmetrical
- The center of mass of the quadrotor coincides with the origin of the body fixed reference frame
- The propellers of the quadrotor are rigid
- The thrust and drag exerted by and acting upon the propellers are proportional to the square of the propeller rotational speed

The vector containing generalized coordinates of the quadrotor can be defined as $q = (x, y, z, \phi, \theta, \psi) \in R^6$. It can be split into two components, for translational and rotational motion. In the inertial frame the translational component of the current center of mass of the quadrotor and the velocities thereof can be expressed as vectors $\xi = (x, y, z)^T$ and $\dot{\xi} = (\dot{x}, \dot{y}, \dot{z})^T$. The roll angle ϕ, pitch angle θ and yaw angle ψ can be combined in a similar manner, in the body frame: $\eta = (\phi, \theta, \psi)^T$, with corresponding angular velocities $\dot{\eta} = (\dot{\phi}, \dot{\theta}, \dot{\psi})$

The transformation matrix $\mathcal{R}$ relating the inertial coordinates to body fixed coordinates can be described as follows:,

$$\mathcal{R}(\eta) = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi) \\ s(\psi)c(\theta) & s(\psi)s(\theta)s(\phi) - c(\psi)c(\phi) & s(\psi)s(\theta)s(\phi) + s(\psi)c(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) \end{bmatrix} \tag{6-1}$$

where, s and c stand for sin and cos operators. Any point of the airframe can be expressed in the Earth fixed frame using -

$$r = \mathcal{R}(\eta) \cdot \xi \tag{6-2}$$

The corresponding velocities can be obtained by differentiation, and the squared magnitude of the velocity at any given point is:

$$v^2 = v_X^2 + v_y^2 + v_Z^2 \tag{6-3}$$

The kinetic energy expression can be extracted from equation (6-3) as follows, assuming a diagonal inertia matrix I:

$$\begin{aligned} T = \quad & \frac{1}{2} I_x (\dot{\phi} - \dot{\psi} s(\theta))^2 \\ + \quad & \frac{1}{2} I_y (\dot{\theta} c(\phi) + \dot{\psi} s(\phi) c(\theta))^2 \\ + \quad & \frac{1}{2} I_z (\dot{\theta} s(\phi) - \dot{\psi} c(\phi) c(\theta))^2 \end{aligned} \tag{6-4}$$

Then the potential energy can be expressed as:

$$\begin{aligned} V = \quad & \int x dm(x)(-g s(\theta)) \\ + \quad & \int y dm(y)(g s(\phi) c(\theta)) \\ + \quad & \int z dm(z)(g c(\phi) c(\theta)) \end{aligned} \tag{6-5}$$

The derived formulas (6-4) and (6-5) can then be combined, using the Langrangian:

$$L = T - V \ , \ \Gamma_i = \frac{d}{dt}\left(\frac{\delta L}{\delta \dot{q}_i}\right) - \frac{\delta L}{\delta q_i}, \tag{6-6}$$

where Γ_i stands for generalized forces. It can be split in two components, for translational and rotational motion:

$$\begin{aligned} F &= \frac{d}{dt}\left(\frac{\delta L}{\delta \dot{\xi}}\right) - \frac{\delta L}{\delta \xi} \\ \tau &= \frac{d}{dt}\left(\frac{\delta L}{\delta \dot{\eta}}\right) - \frac{\delta L}{\delta \eta} \end{aligned} \tag{6-7}$$

The translational force $F = \mathcal{R} \cdot \hat{F}$ depends on the total thrust f, and it has to be adjusted from body-frame into inertial reference frame. The total thrust force $\bar{F}$ is:

$$\begin{cases} \hat{F} = \begin{bmatrix} 0 \\ 0 \\ f \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} f_i \end{bmatrix} \\ f_i = C_T \Omega_i^2; \ i = 1 \dots 4 \end{cases} \tag{6-8}$$

6-4-1 Translational dynamics

Using formula in (6-7), and adding the effects of mass m of the quadrotor:

$$\frac{d}{dt}\left(\frac{\delta L}{\delta \dot{\xi}}\right) - \frac{\delta L}{\delta \xi} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} \tag{6-9}$$

Combining it with the force derived in (6-8):

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \mathcal{R}(\phi, \theta, \psi) \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} f_i \end{bmatrix} \tag{6-10}$$

Using formula in (6-10) gives the following translational dynamic equations of motion:

$$\begin{aligned} \ddot{x} = & \frac{cos\phi sin\theta cos\psi + sin\phi sin\psi}{m} U_1 \\ \ddot{y} = & \frac{cos\phi sin\theta sin\psi + sin\phi cos\psi}{m} U_1 \\ \ddot{z} = & -g + \frac{cos\phi cos\theta}{m} U_1 \end{aligned} \tag{6-11}$$

6-4-2 Rotational dynamics

The Lagrangian for the rotational multicopter dynamics can be written as:

$$\frac{d}{dt}\left(\frac{\delta L}{\delta \dot{\xi}}\right) - \frac{\delta L}{\delta \xi} = \tau \tag{6-12}$$

Where τ stands for the sum of forces due to inputs provided to the system and the gyroscopic effect due to propeller rotation $\tau = \tau^{in} + \tau'$. The inputs are translated into torques acting upon quadcopter by varying thrust of each rotor. This is done by changing the rotor rotational speed Ω_i. These torques can be expressed as:

$$\begin{aligned} \tau_x^{in} = & C_T l(\Omega_4^2 - \Omega_2^2) \\ \tau_y^{in} = & C_T l(\Omega_3^2 - \Omega_1^2) \\ \tau_z^{in} = & C_D(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2) \end{aligned} \tag{6-13}$$

Furthermore, there are additional gyroscopic effects that result from propeller rotation:

$$\begin{aligned} \tau_x' = & J_p \omega_y (\Omega_1 + \Omega_3 - \Omega_2 - \Omega_4) \\ \tau_y' = & J_p \omega_x (\Omega_2 + \Omega_4 - \Omega_1 - \Omega_3) \end{aligned} \tag{6-14}$$

With these effects added, a full set of equations for rotational motion can be assembled:

$$\begin{aligned}
\ddot{\phi} &= \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right) - \frac{J_p}{I_x}\dot{\theta}\Omega + \frac{l}{I_x}U_2 \\
\ddot{\theta} &= \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right) + \frac{J_p}{I_y}\dot{\phi}\Omega + \frac{l}{I_y}U_3 \\
\ddot{\psi} &= \dot{\phi}\dot{\theta}\left(\frac{I_x - I_y}{I_z}\right) + \frac{1}{I_z}U_4
\end{aligned} \tag{6-15}$$

where the system inputs U_1, U_2, U_3, U_4 and Ω are defined as:

$$\begin{cases}
U_1 = C_T(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \\
U_2 = C_T(\Omega_4^2 - \Omega_2^2) \\
U_3 = C_T(\Omega_3^2 - \Omega_1^2) \\
U_4 = C_T(\Omega_1^2 + \Omega_3^2 - \Omega_2^2 - \Omega_4^2) \\
\Omega = \Omega_2 + \Omega_4 - \Omega_1 - \Omega_3
\end{cases} \tag{6-16}$$

where:

$\mathcal{R}$	Rotation matrix
ϕ	Roll angle
θ	Pitch angle
ψ	Yaw angle
Ω_i	Rotor speed
$I_{x,y,z}$	Body inertia
J_p	Propeller inertia
C_T	Thrust coefficient
C_D	Drag coefficient
l	lever

The first input U_1 corresponds to quad-copter throttle and used to increase/decrease altitude by increasing or decreasing thrust of every rotor. Yaw of the vehicle is controlled by inputs U_2 and U_3 that vary the thrust across opposing rotors. U_4 is used to shift the thrust distribution across the rotors to achieve pitching/rolling motion. The quadrotor motion modes are shown in figure 6-2.

There are a number of parameters needed to describe a quad-copter vehicle. It's inertial characteristics depend on moments of inertia I_{xx}, I_{yy}, $I_z z$, rotor inertia I_{rotor} and vehicle mass m. While the mass of the UAV is pretty easy to measure, estimating moments of inertia is less straight-forward and requires a special tool setup. Other relevant numerical values are rotor thrust and drag coefficients C_T and C_D. These are characteristic to rotors used on the UAV and might vary, depending on various factors such as aircraft velocity or rotation speed of the rotor. Rotor arm length l is the distance from the rotor to aircraft centre of gravity and can easily measured directly.

6-4-3 Rotor dynamics

Multicopter drones are typically powered by DC motors. The dynamics of such motors can be described by the following equations:

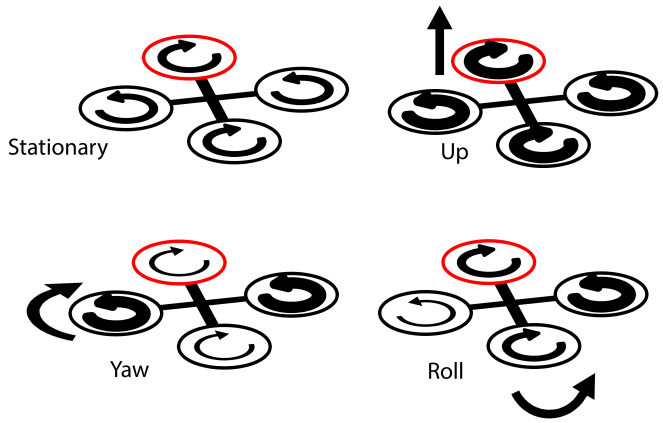


Figure 6-2: Quadrotor motion description. Arrow width is proportional to propeller rotational speed.

$$\begin{cases} L\frac{di}{dt} = u - Ri - k_e\omega_m \\ J = \frac{d\omega_m}{dt} = \tau_m - \tau_d \end{cases} \tag{6-17}$$

Assuming a very low inductance value, the second order dynamics of a DC motors can be approximated:

$$J\frac{d\omega_m}{dt} = -\frac{k_m^2}{R}\omega_m - \tau_d + \frac{k_m}{R}u \tag{6-18}$$

Furthermore, the propeller and the gearbox models can be introduced. Re-writing equation (6-18):

$$\begin{cases} \dot{\omega}_m = -\frac{1}{\tau}\omega_m - \frac{d}{\eta r^3 J_t}\omega_m^3 + \frac{1}{k_m\tau}u \\ \frac{1}{\tau} = \frac{k_m^2}{RJ_t} \end{cases} \tag{6-19}$$

The equation (6-19) can be linearised in the form $\dot{\omega}_m = -A\omega_m + B_u + C$:

$$A = \left(\frac{1}{\tau} + \frac{2d\omega_0}{\eta_g r^3 J_t}\right),\ B = \left(\frac{1}{k_m\tau_t}\right),\ C = \frac{C_D\omega_0^2}{\eta_g r^3 J_t} \tag{6-20}$$

where

u	Motor input
k_e	Back EMF constant
ω_m	Motor angular speed
τ_m	Motor torque
τ_d	Motor load
τ_t	Motor time-constant
R	Motor internal resistance
r	Gear box reduction ratio
η_m	Gear box efficiency

6-5 Active set control allocation

Control allocation is a method that that is applied to distribute the control effort between several controllers when the number of actuators exceeds the number of controlled variables or when various actuators are in conflict with one another. There are a number of solutions available to solve this problem, such as, direct control allocation (Durham, 1994), daisy chainging (J. M. Buffington & Enns, 1996) or constrained linear programming(J. Buffington, Chandler, & Pachter, 1999; Svendsen et al., 2012). In this work, active set control allocation is used (Harkegard, 2002).

The purpose of a control allocator is to compute a set of actuator input commands $u \in \mathbf{R}^m$ than result in an overal control effort, a virtual control $v \in \mathbf{R}^n$, where m represents the number of actuator inputs u, and n is the number of states to be controlled.

The virtual control typically represents the desirable pitch, roll and yaw angular accelerations for conventional aircraft, with vertical acceleration being an additional factor in rotorcraft.

The generated virtual control is represented as Bu. Variable B represents the control effectiveness matrix. The control vector u is typically bounded by upper and lower limits as

$$u_{min} \leq u \leq u_{max} \tag{6-21}$$

In the proposed control scheme the input vector **u** represents the squared motor voltages

$$u = [v_0^2, v_1^2, v_2^2, v_3^2] \tag{6-22}$$

The bounding limits of these values are indicated in table B-1.

The purpose of control allocation is to find such u that satisfies the condition $Bu = v$, given a virtual control command v. In case there are several possible solution, the optimal solution must be selected. And if there is no solution, the value of u resulting in the best possible approximation of Bu must be found.

For an optimum solution the following two conditions can be defined:

$$u_S = \arg\min_{u \in \mathbf{K}} ||W_u(u - u_p)|| \tag{6-23}$$

$$\mathbf{K} = \arg\min_{u_min \leq u \leq u_max} ||W_v(Bu - v_p)|| \tag{6-24}$$

In these equations, **K** stands for the set of feasible actuator inputs that minimize the value of $Bu - v$ (weighted by matrix W_v). A control setting minimizing the $u - u_p$ must be picked. The latter term represents the difference between the desired control input and the possible one, weighted by matrix W_u. These two criteria can be combined to formulate a weighted least squares problem:

$$u_W = \arg\min_{u_{min} \leq u \leq u_{max}} ||W_u(u - u_p)||^2 + \gamma ||W_v(Bu - v_p)||^2, \tag{6-25}$$

where γ is the weighting factor, emphasizing that the prime objective is to minimize the value of $Bu - v$.

The least squares problem that is bounded and equality constrained can be defined as follows:

$$\min_u ||Au - b|| \tag{6-26}$$

$$Bu = v \tag{6-27}$$

$$Cu > U \tag{6-28}$$

where 6-28 is the constraint ensuring that the value of u stays within limits, as stated in 6-21. C is defined as $C = I - I$ and U is defined as $U = u - u$. The goal of an active set algorithm is to solve the posed problem by solving several equality constrained problems in a sequence. During each algorithm step the inequality constrained are considered to be equality constraints, constituting the working set $\mathbf{W}$, while the rest of the inequality constraints are taken out of the equation. The active set of the solution is the working set at the optimum. Algorithm 1 illustrates the application of active set control allocation approach to the problem of quadrotor control, as implemented in the software as `state::control_alloc()`.

Data: Initialize B, u_0, u_{min}, u_{max}, v_0

$$B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & 0 & -1 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

Initialize a feasible starting point u_0 $u_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$

Specify control limits u_{min} and u_{max} $u_{min} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$

$u_{max} = \begin{bmatrix} v_{0,\max}^2 & v_{1,\max}^2 & v_{2,\max}^2 & v_{3,\max}^2 \end{bmatrix}$

Set the initial condition (desired control setting) for the virtual control input v_0

$v_0 = \begin{bmatrix} v_b & v_{02} & v_{13} & v_{0213} \end{bmatrix}$

The quadratic cost function is specified as:

$\|W_u(u - u_p)\|^2 + \gamma\|W_v(Bu - \mathbf{v}_p)\|^2 =$

$$\left\| \underbrace{\begin{pmatrix} \gamma W_v B \\ Wu \end{pmatrix}}_{A} u - \underbrace{\begin{pmatrix} \gamma W_v v \\ W_u u_p \end{pmatrix}}_{b} \right\|$$

Where W_v is the weighting matrix that can be used to allocate priority to certain virtual inputs over others. Altitude control is given the highest priority, while yaw control is given the lowest. The weighting matrix W_u can be set as identity matrix $W_u = I$.

Solve

$u_W = \arg\min_u \|Au - b\|$

$u_{min} \leq u \leq u_{max}$

Specify d as $d = \begin{pmatrix} b - Au \\ b - Au \end{pmatrix}$

while *Max number of iterations allows* **do**

  Use solver to find the solution to $A_{free}p_{free} = d$

  Replace the values $p \leftarrow p_{free}$

  Update $u_{opt} \leftarrow u_{opt} + p_{free}$

  Check that the limits($u_{min} \leq u \leq u_{max}$) are satisfied and find the number of infeasible solutions if not

  if *infeasible solutions present* **then**

    Find the lowest distance from the limit among the free variables and remove from the pool

    Denote the lowest distance as α and update the input vector

    $u \leftarrow= u + \alpha p$

    Update d

    $d \leftarrow d - \alpha A_{free}p_{free}$

  end

  else

    return *The optimal solution* u_{opt}

  end

end

Algorithm 1: Active set control allocation application

The applied control allocation algorithm allows to resolve possible conflicts between various controllers and to ensure that all inputs remain within specified limits, while prioritizing more important control modes(altitude hold) over less important ones(yaw).

6-6 Drone control scheme

The full drone controller can be split into two parts: the inner loop, the and outer loop control. Lower-level inner loop controller is designed to track reference pitch, roll, yaw and altitude signals. These dynamics are basic double integrator-type systems, although the simulated model also includes motor dynamics, that introduce some higher order dynamics in the form of a simulated thrust response lag. Higher-level outer loop controller is designed to track a position reference, such as x and y coordinates of the vehicle, and perform maneuvers. The outer loop controllers pass on reference signals to the inner loop, and they are set up in a cascaded fashion. An example of a basic drone control scheme layout is shown in Figure 6-3. There are four inner-loop controllers in place: roll, pitch, yaw and altitude control. There

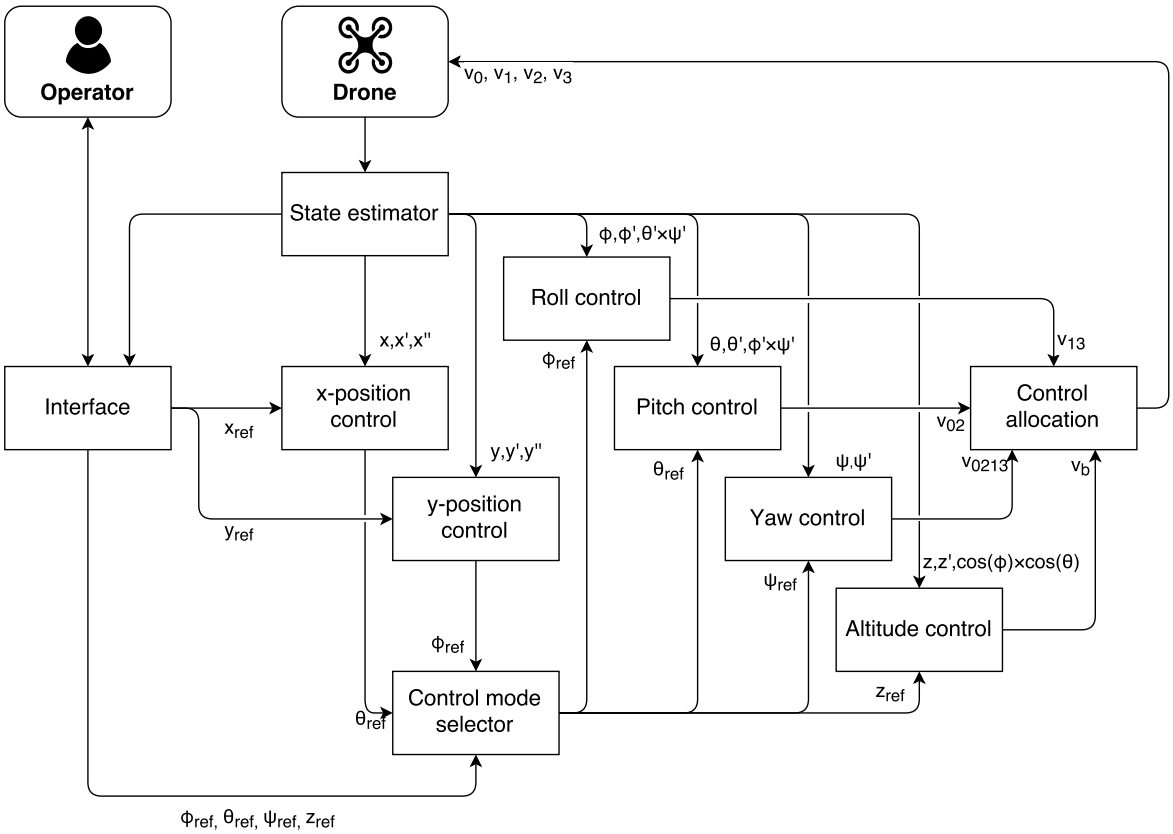


Figure 6-3: Drone control scheme

are also two outer loop position controllers, for states x and y. The operator interacts with the controller via an interface. Reference signals for pitch and roll φ_{ref}, θ_{ref} can be either set directly through the interface, or as a command from higher-level x and y controllers. Yaw angle and altitude reference signals ψ_{ref} and z_{ref} are set directly. It is also possible to add an additional layer of controllers, that combine flight heading and yaw control or performs some set of maneuvers, for example. The four inner-loop controllers produce scaled control inputs v_b (altitude), v_{02} (pitch controller), v_{13} (roll controller), and v_{0213}. Inner-loop controller inputs and outputs are defined as follows:

Controller	Inputs	Outputs	
Altitude	z_{ref}, z, $\dot{z}$, $c(\phi)c(\theta)$	v_b	$= v_0^2 + v_0^2 + v_0^2 + v_0^2$
Roll	ϕ_{ref}, ϕ, $\dot{\phi}$, $\dot{\theta} \times \dot{\psi}$	v_{13}	$= v_1^2 - v_3^2$
Pitch	θ_{ref}, θ, $\dot{\theta}$, $\dot{\phi} \times \dot{\psi}$	v_{02}	$= v_0^2 - v_2^2$
Yaw	ϕ_{ref}, ϕ, $\dot{\phi}$, $\theta \times \psi$	v_{0213}	$= v_0^2 - v_1^2 + v_2^2 - v_3^2$

Note that reference outputs still have to be processed to produce usable motor voltage inputs $v_0 \dots v_3$. Each voltage is limited, and different controllers might be in conflict with one another. This makes it necessary to include a control allocation module, in order to interpret the resulting voltage differentials as usable motor inputs.

6-7 Discussion

This chapter had dealt with dynamics and control methods of UAV drone. Various model identification methodologies were described. Following, a description of a mathematical model describing quadrotor dynamics was given, as well as methodology for implementing a computer simulation of this model. Quadrotors can be controlled using a set of four actuators(rotors), that are responsible for pitch, roll, yaw and altitude control. These controls can be combined to add another set of control modes, such as positional (x-y axis) control. A full control scheme dealing with all these modes was proposed. In addition, the control allocation problem was discussed. Drone controls are coupled with one another and are subject to limitations. Active set control allocation method was described, designed to overcome these issues. A model inversion-based controller based on RBF networks and polynomial curve fitting was described as well.

Chapter 7

Reinforcement-learning based control of the UAV

Previous chapters have outlined some topics and techniques, relevant to UAVs, control of such vehicles, reinforcement learning, and other related topics. This chapter deals with design and implementation of a full controller of a UAV, such as described in. Two techniques are considered: a conventional PD and a reinforcement learning-based controllers. The PD controller performance is used as a benchmark for RL-based approach, and it is used to validate the resulting controller. The PD gains are optimized using the PSO algorithm, described in 5-1. The RL-based controller is based on Q-learning, described in chapter 3. Both controllers incorporate an adaptive model-based inversion dynamics inversion scheme to generate actuator inputs. In this chapter, the design of both types of controllers is discussed, and the control efficiencies are compared. Several controllers are designed, governing the pitch, roll, yaw and vertical motion dynamics. A schematic of how these controllers are combined to achieve full control of the UAV is shown in Figure 6-3

7-1 Controller layout

A general UAV control scheme was described in 6-6. The layout of the control scheme consists of two parts: the inner and the outer control loops. Two types of controllers are developed to fill the role of inner-loop control of the UAV: a conventional PID and a reinforcement learning approaches.

Figure 7-1 illustrates a conventional feedback PID controller. It processes the signal reference offset $e(t) = x - x_{ref}$. The absolute value of this offset is multiplied with the proportional gain K_p, the integral of this error is multiplied with K_i and the rate of change of this error is multiplied with gain K_p. The output of the PID controller is the desired acceleration $\ddot{x}_d$. The model inversion module is then tasked with generating a suitable virtual input to achieve this desired acceleration.

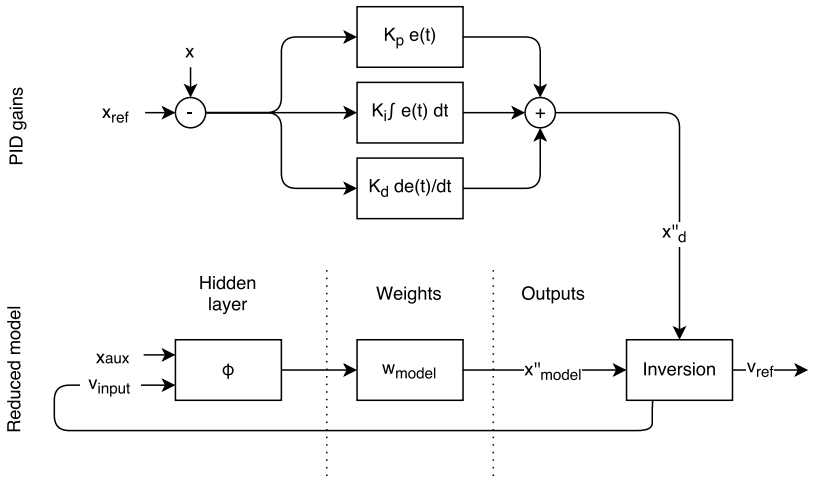


Figure 7-1: PID control scheme

PID controllers are relatively easy to implement, and are sufficient for most real-world control applications. It can serve as a validation for reinforcement learning-based controller behavior.

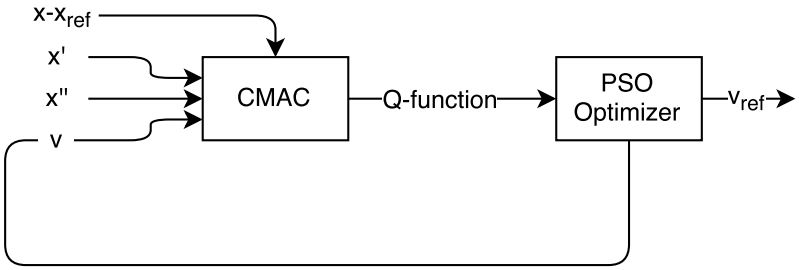


Figure 7-2: CMAC based RL control scheme

The proposed Q-learning controller is more complex than the PID one. Two approaches were tested when designing the RL controller. One using the CMAC net, and another one using the RBF neural net. The first approach used CMAC generalization to store the Q-function. There is no analytical solution to CMAC generalization function, so in order to optimize its output the PSO algorithm was used, as described in 5. However, this approach proved not viable. The controller failed to learn the policy. One of the problems with it was the phenomena that was described in 4-3. The policy typically failed to converge, and when it did the behavior was unacceptable.

Another way to implement the policy is to store it in the form of encoded RBF neural net, as described in 4-1. Like with the PD controller, x, $\dot{x}$ and $\ddot{x}_{ref}$ are used as inputs. The difference comes from how the desired control inputs are produced: in a reinforcement learning controller the actions are generated by maximizing the output value of the network.

The policy is stored in the form of an RBF neural net that accepts at least three inputs: the offset between the current and the reference states $\Delta x = x - x_{ref}$, current state derivative $\dot{x}$ and the desired state acceleration $\ddot{x}_d$. The variable $\ddot{x}$ defines the action that can be taken by the agent. When states Δx and $\dot{x}$ are sampled at some point in time the action variable $\ddot{x}_d$

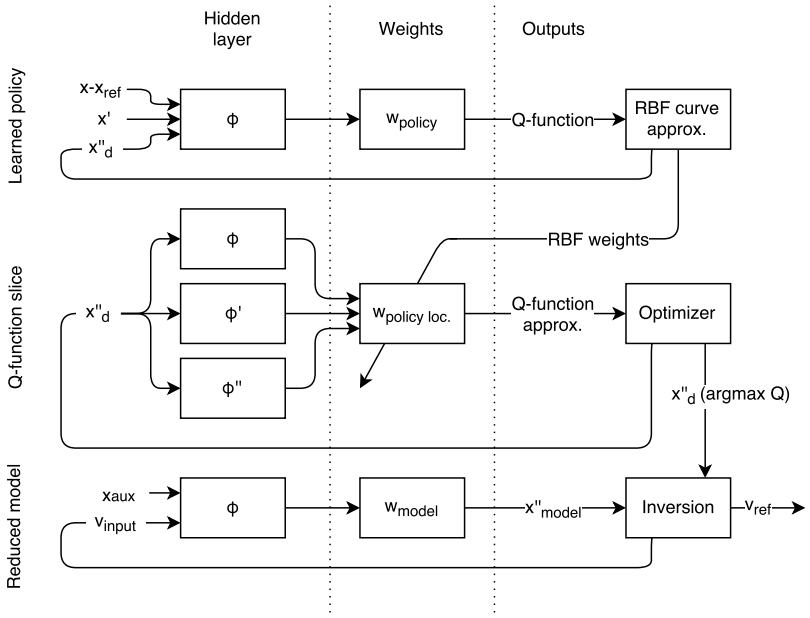


Figure 7-3: RBF net based RL control scheme

becomes the only unconstrained input of the generalized Q-function. Therefore it becomes possible to make an instantaneous "slice" of the Q-function, which can be described as a one-dimensional RBF-based curve. In order to do this, the output of the neural net that describes the Q-function is sampled at variable action values $[x_{d,0}, x_{d,1}, \ldots, x_{d,N}]$. The outputs $Y = [Q_0, Q_1, \ldots, Q_N]$ are stored and used to generate a one-dimensional RBF-based function, using the procedure, described in Section 5-2-2.

7-2 Reduced model and dynamics inversion control

The complete UAV dynamics, outlined int the previous section, consists of six states describing drone position and attitude(x, y, z, ϕ, θ, ψ), four rotor speeds (ω_0, ω_1, ω_2, ω_3) and four virtual inputs (v_b, v_{02}, v_{13}, v_{0213}) that can be translated into direct motor voltage inputs (v_0, v_1, v_2, v_3). All motions that the UAV goes through, along or about the x-axis, y-axis or z-axis are cross-coupled with other motions of the aircraft. Pitch, roll and yaw dynamics are all inter-connected, while the translational motions are connected to the current aircraft tilt offset from the vertical z-axis. This relatively complex model can be reduced into a more simple one. This simplified model could serve a double purpose: one is to serve as an aide in designing an inversion-based controller that is capable of generating virtual inputs, based on the desired state, another one is to be used for off-line training of the reinforcement learning algorithm.

Combining these two functions of the model allows to designate the desired state as the policy action. This offers some advantages, compared to a policy that produces a virtual input directly. The main advantage is that if the model used for off-line training deviates from real-world dynamic behavior this has less of an impact on learned policy validity: the policy does not have to be re-learned if the virtual inputs result in different accelerations using

software in the loop(SIL) or hardware in the loop(HIL) training. Learned state transitions, or a "path" taken on the way to achieving the goal, should remain relatively unchanged regardless. Only the reduced model itself would have to be adjusted to better match the real-world dynamics. In essence, this would result in an adaptive controller, where the model is dynamically updated to match the real-world dynamics.

Using a reduced model offers another important advantage: it allows to reduce the order of the system, and this greatly simplifies the structure of the policy itself. Reducing the number of inputs results in a smaller state-space, allowing the policy to be learned more quickly. Additionally, generalizing a function that has many inputs requires more parameters to be set, compared to a function with relatively few inputs. A simpler function can be evaluated more efficiently, reducing the computational load.

This approach also presents some disadvantages, compared to training on-line or using a model of the process that describes the dynamics more fully. The resulting policy would only be valid for the reduced model of the process, and depending how closely this model matches the process, it might result in an inviable control solution. Also, the resulting policy would not take any of the complex relationships between different states into account. Of course, in real operation these effects would take place and will have an effect on controller effectiveness.

Some of the dynamics only play a marginal role in the overall performance of the controller. This includes the response delay of the actuators(rotors) and the cross-coupling modes. The delay between the change of voltage, supplied to the rotors, and the response of the system is very slight. It can safely be neglected for a large enough time step. While the cross-coupling effects should largely be overshadowed by the contribution of control actuators during moderate flight maneuvers. More aggressive maneuvering might result in a larger deviation.

Any cross-couplings, present in the pitch roll and yaw modes, are assumed to have no influence, and applying a virtual input to the system results in an immediate change of state, without any delay. The full dynamics model is split into four simpler sub-models, describing pitch, roll, yaw and altitude motion. An RBF-net is used to store these dynamics. Each RBF net maps the relationship between virtual inputs and output acceleration of the vehicle. This relationship is nearly linear for a sufficiently large step-size. However, in reality the steady-state response of the system is not linearly related to scaled inputs. Estimation of the simplified model of the process is done by applying a range of inputs to the full model, performing a prioritized state sweep, and recording it's responses over a sufficiently large time-step. The results are fitted using an RBF net. When using a real system the same methodology can be applied, performing a prioritized sweep of the input-output state space while isolating it from the cross-coupling dynamics. The model structure is demonstrated in figure 7-4. An example of the model output is shown in Figure 7-5, for the pitch control mode.

This model can be easily inversed by using polynomial approximation to the RBF net and then finding it's roots. An array of virtual input samples $[v_0, v_1, \ldots, v_N]$ is supplied as input to the model and the outputs $Y = [\ddot{x}_{m,0}, \ddot{x}_{m,1}, \ldots, \ddot{x}_{m,N}]$ are recorded. A local polynomial curve approximation is generated every time a new virtual control input is requested, based on supplied desired second state derivative $\ddot{x}_d$. This curve can be expressed as

$$Y = A\theta, \tag{7-1}$$

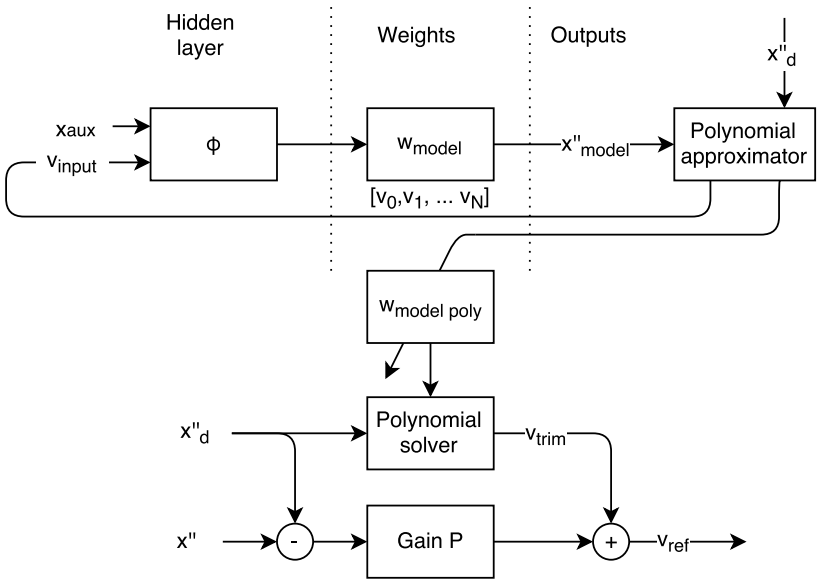


Figure 7-4: Inversion controller

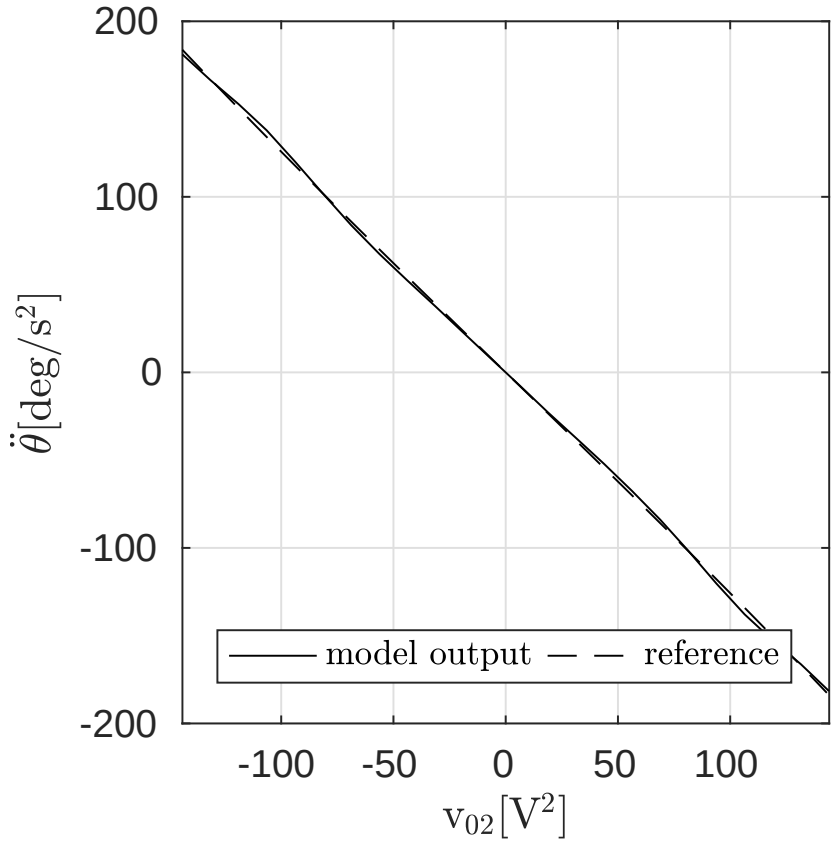


Figure 7-5: Reduced dynamics model and inverse controller

similar to the procedure, described in 5-2-1. The parameter vector θ describes the polynomial that follows the shape of the neural net that stores the model. In essence, the model itself

describes the trim state voltage input for any given virtual input value, denoted by x_{input}. In order to reverse this model, the parameter vector θ is modified by adding the desired acceleration $\ddot{x}_d$ to it's first term, that describes the polynomial bias. Given a parameter vector θ that approximates the reduced model

$$\theta = \begin{bmatrix} p_0 & p_1 & \dots & p_N \end{bmatrix}^T, \tag{7-2}$$

the adjusted parameter vector θ^* becomes

$$\theta^* = \begin{bmatrix} p_0 + \ddot{x}_d & p_1 & \dots & p_N \end{bmatrix}^T. \tag{7-3}$$

The roots of the resulting system are found by finding eigenvalues of the companion matrix A^*

$$A^* = \begin{bmatrix} -\frac{p_1}{p_0+\ddot{x}_d} & -\frac{p_2}{p_0+\ddot{x}_d} & \dots & -N\frac{p_N}{p_0+\ddot{x}_d} \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \tag{7-4}$$

$$r = eig(A^*) \tag{7-5}$$

These roots r correspond to trim state values of the virtual inputs for a given value of x_d. Due to extra dynamics taking place if this value was applied directly, the system might start moving towards the desired goal, but the equilibrium might be achieved before it actually reaches it. In other words, the controller would have a severe undershoot. In order to account for this discrepancy a small proportional gain is applied. The full inversion controller scheme is illustrated in the bottom part of Figure 7-4. The resulting inverse model state controller performance is illustrated in figure 7-6 for pitch-mode control.

Figure 7-6 shows that the inverse controller is capable of closely adjusting the motor input differential v_{13} to match the reference acceleration $\ddot{\theta}_{ref}$. Note that the transitions happen very quickly, so for the reduced dynamics these dynamics can be assumed to happen instantaneously with sufficiently large time step.

7-3 Controller design

The initialized controller has several parameters that have to be set in advance: neural net generalization function parameters, such as the supplied states, the number of neurons per-dimension, and any auxiliary states that might affect the dynamics that have to be controlled. Also, there are Q-learning trial and value function parameters: duration of one episode, value function update rate α, discount parameter γ, trace decay rate λ, and the duration of applied trial action.

The trial length, discount rate, and action duration length are critical parameters: control dynamics such as pitch and roll have a much shorter response time, compared to yaw control

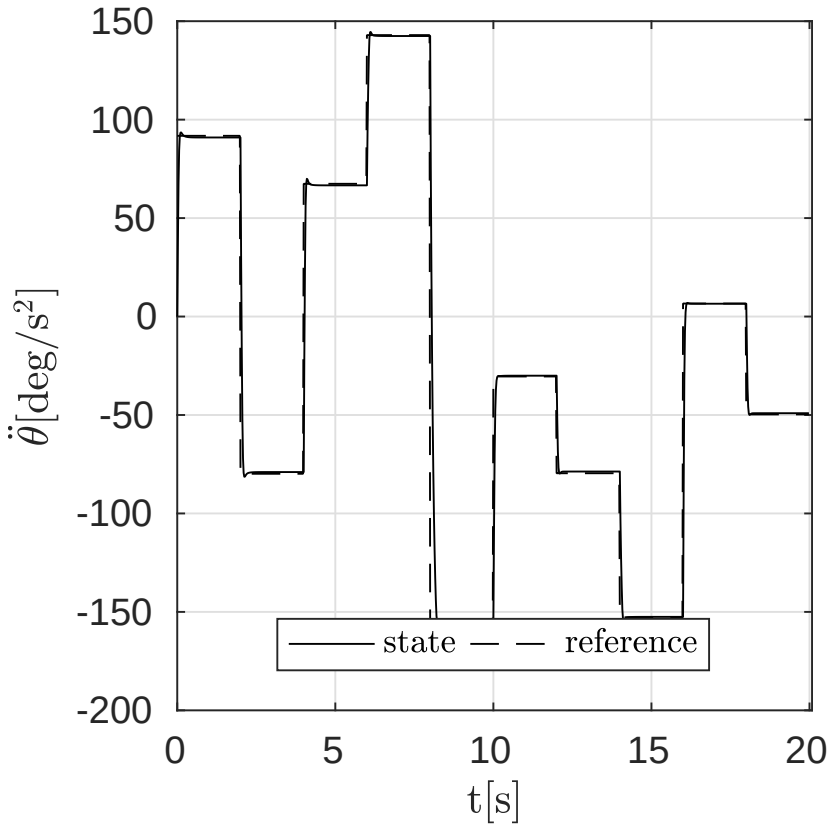


Figure 7-6: Time series of inverse controller test run

for example. This means that the time horizon over which an episode has to be run must be shorter, and the inputs have to be adjusted more rapidly.

Another learning parameter that has to be set is the reward function. There are two basic ways to do it: assigning a fixed penalty at each time step with the magnitude depending on whether the goal is reached or not. This is hard to achieve with continuous states if the goal is not just some threshold to be passed, but a reference to be tracked for a period of time. Another way is to use a reward function, that changes depending on current system state. There are various ways to set up this function. The way the reward is assigned can influence the learning process during the exploration stage, as well as the overall behavior of learned policy. For example: if the reward is always positive, then the controller might give preference to actions that had been explored previously, despite the possibility that some of the un-tried actions might lead to higher value. Likewise, assigning negative rewards results in preference given to actions that had not been explored previously, despite the fact that the optimum action already had been tried. In effect, this works the same as pre-setting the value function. Another factor when designing a reward function where several states are combined is the maintenance of balance between different states. For example, a reward function that assigns a lower penalty for lower state error but does not take state derivative into account might result in high overshoot. On the other hand, a reward function that assigns a high penalty for converging "too fast" might also result in a controller that has slower response time or poor steady-state performance.

7-4 Policy training

It had been shown that the computation time for an RL algorithm could grow exponentially (Whitehead, 1991) depending on the number of states. However, it was also demonstrated that using more efficient exploration techniques this time could be reduced to rise polynomially (Carroll, Peterson, & Owens, 2001), so exploiting effective exploration techniques is paramount when dealing with a highly complex model.

Policy learning process progresses in several stages. First, a conventional controller is applied to a simplified model of the system. This controller is designed for optimal hysteresis control. The model used does not contain any higher-order dynamics, such as the lag introduced by motors. It is assumed that any inputs applied to the system act instantaneously. Several simulated trials are executed, using initial randomized conditions and reference control states. Time series data of system behavior in response to such a controller is used to train a prime policy, that does not take into account any couplings between different control states. Using this approach allows to approximate a near-optimal policy in a very short time.

After the first stage is completed, the conventional controller is disabled and a more complex model of the process is used (e.g. one that includes rotor dynamics). The value function generalization function is extended to include auxiliary states, to make it adaptive. More simulated trials are executed. The learning progresses as follows: during initial trials, the model states are initialized close to the goal. This way the first thing that the controller "learns" is how to maintain trim state reference tracking. Once the desirable steady state error falls below a pre-set threshold, the initial conditions are moved further away from the goal, forcing the controller to find a policy that describes the transition from any given initial state to a goal state. This process continues until the entire desired state-space range had been covered, after which more trials are executed with initial conditions set randomly throughout the search space.

In the third stage the controller is applied to real world, e.g. it is used as a primary means of control in a UAV drone. Again, several training runs are executed, starting with the trim state and exploring the search space. This is the stage that takes the longest time-wise, but it also produces the most accurate results. Any discrepancies between the UAV flight dynamics and the model used are evened out by using real world data to update the controller.

7-5 Controller policy design

The controller policy is stored as an RBF neural net. Each controller neural net is initialized with varying inputs. Each state is defined by some parameters. These parameters include:

1. State id
2. State reference
3. Scaling function type
4. Resolution
5. State limits

State id stands for the location of the state in the state vector. The states could either be raw values of the current system state, input magnitudes or a product of a combination of states, defined as auxiliaries. The state reference specifies the way states are put into the net: this reference could either be absolute, meaning the absolute value of the state, or it could be defined as the magnitude of the difference between the current system state and system reference set point. The resolution specifies the size and number of neurons per-state. The scaling function type specifies the type of pre-scaling, applied to neural net inputs. This could either be linear mapping, power or an exponential scaling. The state limits specify the region in which the network approximator is active. The individual variable resolution parameter is corresponding to the amount of generalization applied by the function approximator.

A policy consists of several grids added together. Additional policy parameters include:

1. Action state id
2. Reference state id
3. Reward function specifications

The action state is the used action state. The control state is the state being controlled. The reward function specifies the reward that the agent gains, depending on its state transitions. The reward could be defined in a variety of ways. It is typically higher when the agent state is close to the reference state. The rewards could be either positive or negative, which has an impact on agent behavior, at least during initial learning stages. Assuming the Q-function is initialized at 0, updating it with a positive reinforcement after each episode iteration results in a high value assigned to the action taken, compared to all other possible actions that are assigned a zero value at initialization. As a result, the agent will tend to select actions that it had applied previously for as long as the outcome of other actions is still uncertain. This heuristic is called an optimistic approach. As a result of using it, a substantial negative evidence is needed to eliminate an action from consideration (Kaelbling et al., 1996).

7-6 Controller performance

This section discusses the resulting controller performance and compares it to conventional(PID) controller. The performance of each controller is evaluated using the following reward function:

$$r = \dot{s} - sign(\Delta s) * \sqrt{sign(\Delta s) * \Delta s} \tag{7-6}$$

Variable Δs represents the difference between controlled state s and the reference state $\Delta s = s - s_{ref}$. The reward function is illustrated in figure 7-7 for the pitch controller.

7-6-1 PID controller

In order to make a fair comparison, the PID gains are tuned using PSO optimization. The reward function is used to evaluate controller performance. The algorithm is shown in 2.

Data: Initialize $\bar{P}$, $\bar{I}$, $\bar{D}$ arbitrarily
Define objective function $V = \sum_{step=0}^{N} r(s)$ for an evaluation episode spanning N steps.
while *current iteration* $<$ *iteration limit* **do**
  for *each gain set* P_i, I_i, D_i **do**
    Update particle velocity v_j^{k+1} using (5-1)
    Update particle coordinate $[P, I, D]_i$ using (5-2)
    Substitute P, I, D gains to currently selecting ones
    Run an episode supplying randomized inputs to calculate cumulative reward V_i
    if $V_i > V_{i\,\max}$ **then**
      Update current particle optimum value and particle coordinate
      $V_{i\,\max} \leftarrow V_i$
      $P_{i\,\max} \leftarrow P_i$
      $I_{i\,\max} \leftarrow I_i$
      $D_{i\,\max} \leftarrow D_i$
    end
    if $V_i > V^*_{i\,\max}$ **then**
      Update current swarm global optimum value and particle coordinate
      $V^*_{\max} \leftarrow V_i$
      $P^*_{\max} \leftarrow P_i$
      $I^*_{\max} \leftarrow I_i$
      $D^*_{\max} \leftarrow D_i$
    end
    if *solution converged* **then**
      break
    end
  end
end
return $[P^*_{\max}, I^*_{\max}, D^*_{\max}]$

Algorithm 2: PID optimization algorithm

7-6-2 RL-based controller

This section outlines the training procedure and specific configuration of the Q-learning controller. The training process begins with the preliminary generation of several potential policies. The training progresses for as long as the following updates generate an improvement in evaluation score. The evaluation is done by performing a test run with the model while the controller is enabled and recording the average cumulative reward. The test signal consists of a mix of step and sinusoidal inputs. Step input allows evaluating steady state performance of the controller, while the sinusoidal signal is designed to assess the response of the controller to a moving reference.

Several policies are generated, and the best one is selected. The results of this preliminary stage are illustrated in figure 7-8 for the pitch controller.

It is notable that there is a considerable spread between various policy scores. The policy quality starts to decrease after a few initial test runs. Therefore it is important to be able to detect it and prevent it from getting worse as the learning continues.

Data: Initialize the policy
Define reward function $r = \dot{s} - sign(\Delta s)\sqrt{sign(\Delta s)\Delta s}$
while *current iteration < iteration limit* **do**
  Reset policy approximation function weights
  while *new score > previous score* **do**
    switch *rand() % 3* **do**
      case *0* **do**
        Run a randomized episode series
      end
      case *1* **do**
        Run a state sweep episode series
      end
      case *2* **do**
        Run a variable signal episode series
      end
    end
    Calculate the new score
    if *Current score > best score* **then**
      Store current policy as the best one
    end
  end
end
Load the best scoring policy
while *true* **do**
  switch *rand() % 3* **do**
    case *0* **do**
      Run a randomized episode series
    end
    case *1* **do**
      Run a state sweep episode series
    end
    case *2* **do**
      Run a variable signal episode series
    end
  end
  Calculate the new score
  if *Current score > best score* **then**
    Store current policy as the best one
  end
end

Algorithm 3: Policy search algorithm

There are three training methods used to train the policy: random state sweep, grid-based state sweep and variable signal response. In a random state sweep, the initial state is placed randomly, close to the final goal state. As the learning progresses, the initial state is placed further and further from the goal. This approach allows to quickly generate a policy valid near the goal region and then refine it by moving the initial position further away. During the

grid-based search, a sweep of the entire state-space is performed. This strategy is designed to cover the entire state-space and to cover the states that might not have been visited during a randomized search. Variable signal training is done by letting the simulation run continuously and varying the reference set point(goal). This is designed to better simulate the actual conditions of the system in operation. During the training stage, the strategy is selected at random.

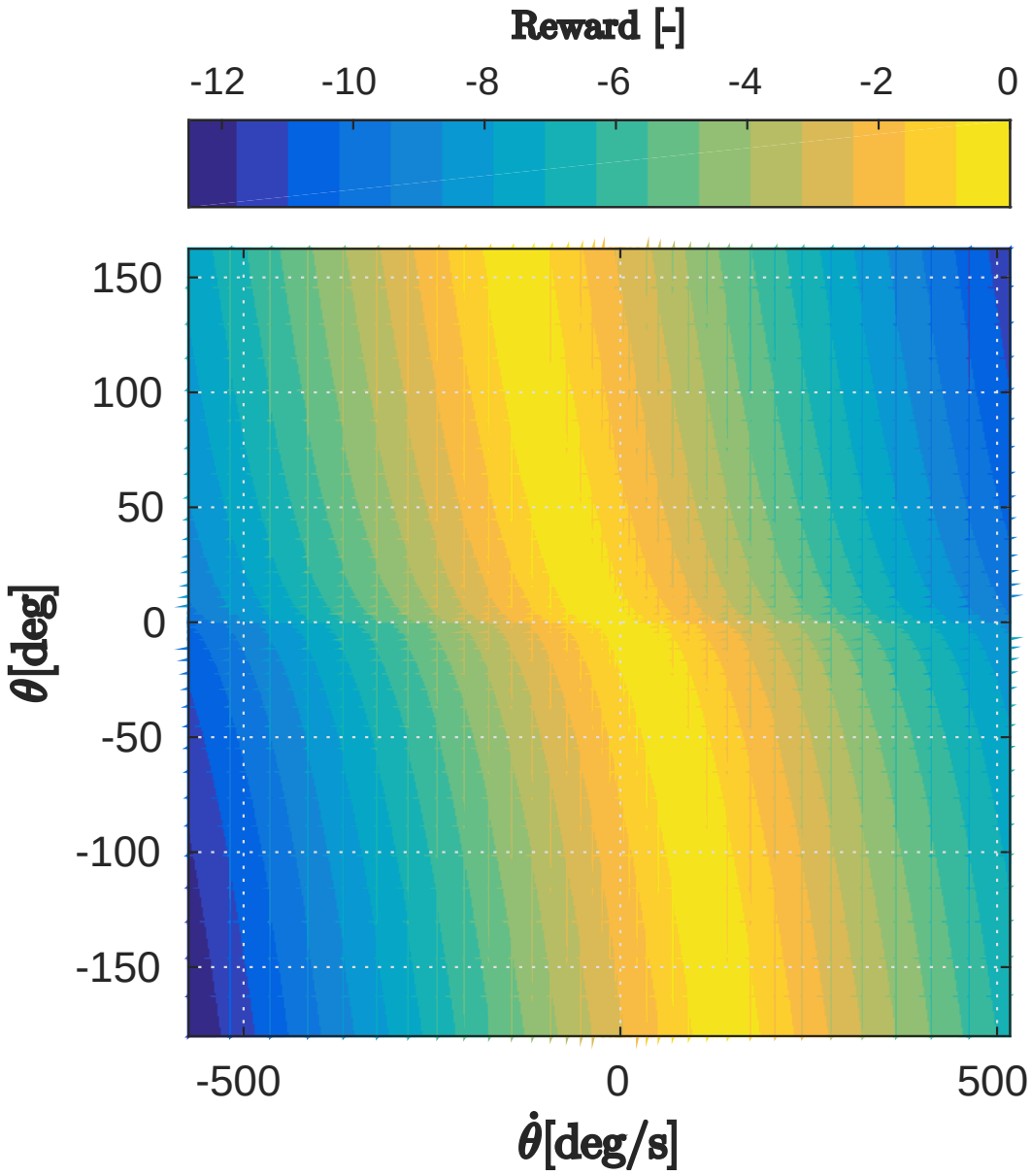


Figure 7-7: Pitch controller reward function

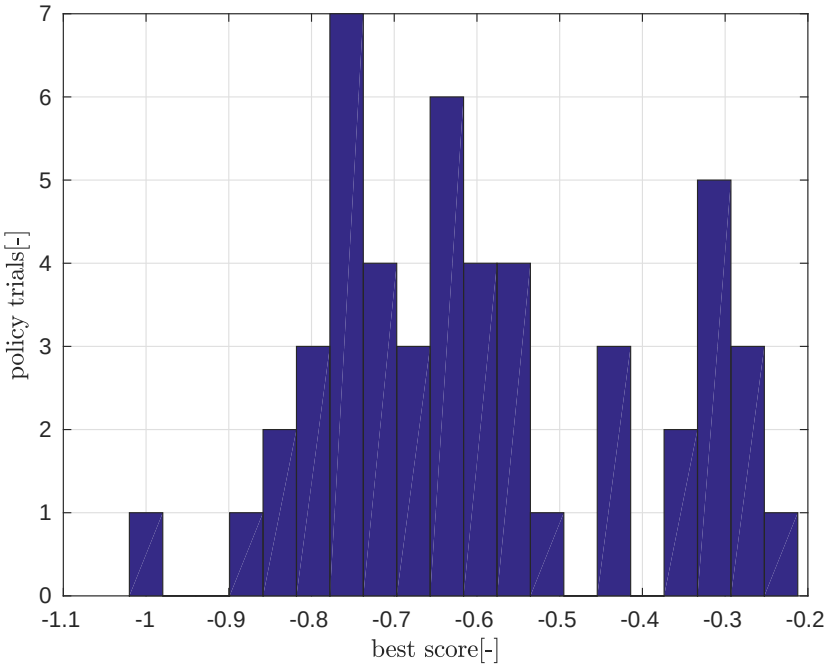


Figure 7-8: Score distribution of preliminary policy search

Chapter 8

Conclusions

This thesis is a contribution towards applying reinforcement learning methodology to control problems, that are typically approached using conventional techniques, e.g. PD control. A continuous state and action reinforcement learning framework was applied to a simulated quadrotor model. The performance of the resulting algorithm was compared to that of a conventional controller. This chapter summarizes some of the background, describes the framework and some of the results. It also suggests direction for further research.

8-1 Summary

Reinforcement learning methodology is being used in increasingly more fields, including aeronautics and UAV control. Chapter 3 outlined some of the current state of art research in the field of reinforcement learning, the basic theory behind it and introduced the Q-learning algorithm and a few of its derivatives. In the chapter A an experiment was proposed. The experiment consists of a simulated multicopter drone, trained to follow a reference signal for reference pitch, roll, yaw and altitude. The experiment chapter outlined the software and its modules, and how they interact with one another. The software was implemented in C++.

Originally, Q-learning was designed to deal with discrete state systems. Whereas the real world robotic systems have continuous states and actions. Therefore, the Q-learning algorithm has to be adapted to the purpose. The Q-function must be generalized. Chapter 4 describes some generalization methodologies: the neural networks and the CMAC net. Some advantages and disadvantages of each method are discussed. A novel hashed neuron net algorithm is proposed that combines the features of the radial basis function neural networks and CMAC encoding to increase the computational efficiency of the algorithm, allowing it to function in real time.

In order to calculate the optimum action, the learned Q-function must be optimized. Chapter 5 introduced the Particle Swarm Optimization (PSO) method and an approach, based on curve fitting with linear regression. The PSO method can work with any type of function, but it is more computationally expensive, compared to curve fitting.

Chapter 6 introduced a mathematical model of the drone. A brief overview of various modeling methods were given, followed by a set of equations of motions of the UAV. This model was used to train and test the reinforcement learning framework. Some notes were given on the implementation of the model in the experiment software, e.g. the numerical solver procedure used to run the simulation. The control scheme was described, with outer and inner loop controllers acting to achieve full control of the drone. A control allocation scheme was outlined as well. It is necessary to implement because otherwise the system control inputs might be in conflict with each other because of actuator input over-saturation.

Since running the full model of the process might not be practical in reality, for example due to some discrepancies between the mathematical model and the real dynamics, a reduced order model is constructed from the primary one by using system. The reduced model consists of a neural network, trained using flight trajectory data and system inputs. Using it is meant to shorten the training time, and it can easily be adapted if the parameters of the real system have changed. The same model is used to implement a dynamics inversion controller, capable of calculating actuator input based on reference acceleration state.

Chapter 7 described the controller layout in more detail. It specified the inputs and the outputs of the control scheme. And how they are processed and passed on to Q-function optimization and the inverse model control to generate actuator inputs. Some of the learning parameters and the reward function were discussed as well. A brief overview of experimental results was given as a conclusion.

A practical continuous state, continuous action $\mathcal{Q}$-learning controller framework has been described and tested using a model of a multi-copter. This work demonstrates that RL methodology can be applied to the inner-loop control of UAVs. The described approach combines model identification and offline learning using a reduced order model of the plant. An optimized hashed neural network algorithm used to store the Q-function values allows to optimize the computational load of the algorithm, making it suitable for online applications. The performance of the algorithm was validated and compared against that of a conventional proportional-derivative controller and was found to exceed it.

The algorithm was found to be capable of generating a working policy, but the policy itself is still inherently dependent on learning parameters, e.g. the reward function structure. It is difficult to express the more subjective desirable performance characteristics (such as the controller responsiveness or smoothness of action) as an objective numeric function. A lot of parameter tuning was required to achieve desirable behavior from the controller. Selection of an appropriate reward function was found to be crucial to correct behavior of the controller.

In addition, the algorithm has shown to be quite inconsistent in training optimal policies for any given trial run: after a certain training period the policy could either converge to a non-optimal solution or get over-written and deteriorate after reaching the near optimum. Therefore several policies had to be generated simultaneously and evaluated to find the best one. This problem is not unique to offline-based training methodologies. If the experiment was carried out with a real drone this would introduce a lot of difficulty in training process, since the progress would have to be reset every time and a new policy would have to be learned. With an offline approach it's not an issue because several learning trials could be executed in parallel in a very short time.

8-2 Future Directions

The direction of future research would include testing the resulting algorithm in an experimental setting that involves a physical system(a multi-copter drone). For this the framework would have to be adapted to communicate with the drone via a data link, receiving the flight telemetry and transmitting the commands. The algorithm itself is computationally expensive and might not be able to function on a typical embedded processing unit, installed on commercial drones. It needs a dedicated processing unit with enough power. Nevertheless, since the computers are getting smaller and cheaper, it can be adjusted to run on a single board computer, such as Raspberry Pi.

In this work only the inner loop dynamics were considered. The control scheme can be extended to incorporate the outer loop control as well, using the same methodology. The model can also be adjusted online to achieve a more accurate estimation of the plant dynamics and making it adaptive. Furthermore, the multicopter model used behaves like a linear system. The developed algorithm could also be applied to highly non-linear systems, where achieving the goal would require executing several actions in a sequence. These are the types of systems for which the reinforcement learning shows the most promise. With the outer loop implemented, it is possible to train the drone to perform complex maneuvers, which would otherwise be impossible to achieve using only PD control scheme.

Furthermore, one of the objectives of this framework was to combine online and simulated offline training methodologies. This approach allows to decrease the risks associated with live testing of a robotic system and to decrease the amount of human supervision required. Safety management is a crucial problem when performing live tests: the system needs a lot of time to learn, the trials have to be run repeatedly, and in the early stages of the learning process the agent might act unpredictably. The framework still requires at least some initial data from the live tests, in order to build a reduced model of the process. An safety filter may be implemented to guarantee that the system would avoid the unsafe situations during the initial learning stages, and a filter could also be applied to the policy itself, resulting in a system that not only acts optimally, but also actively avoids danger.

Appendix A

Experiment Software

The experimental set-up of this research consists of the creation of a computer program, to develop and test the intelligent control approach. The program has to be able to "learn" control strategies, and it also has to be able to simulate a drone to apply them. In the later stages of the project simulated environment would be replaced by a real vehicle. A communication interface has to be implemented to integrate with the drone software and use the learned policy. The general architecture of the program is shown in figure A-1.

The program consists of several components, implemented in C++ and using Qt framework for the front-end output. The *policy engine* is the main module of the program. It is responsible for taking user inputs and passing commands to the UAV using a *wireless link*. The inputs themselves consist of joystick controls(stick position or buttons pressed) and *policy search parameters*, derived from values supplied to GUI interface or from configuration files. The policy engine executes several *simulated learning* process threads. Each of them contains an instance of a learning algorithm, as described in chapter 3. All cases of the policy search algorithm are using the same *drone dynamics* model, but each of them applies excitations to different inputs(such as pitch/roll/yaw control, or supplying a goal state value to an already learned controller). Also, each policy search process observes a different set of outputs, depending on what the controller goal states are, and what vehicle dynamics are most affected by the controller. The learned policy is then generalized and stored in the form of *RBF Neural Network*.

The nature of the program requires it to be operable in real-time, with several tasks being executed simultaneously. It is important to ensure that the communication between various program modules does not lead to any conflicts, such as race conditions or read/write conflicts. For this reason, a *task management* module and a *thread-safe data access* buffers have to be implemented as well.

Model identification module observes input and sensor data on the board of the drone and identifies the drone dynamics, producing an emergent model as the learning process progresses. This model is used by the policy engine to run simulated episodes. In this way, the drone is not required to try and execute any actions to learn a policy. It is done by the software, reducing

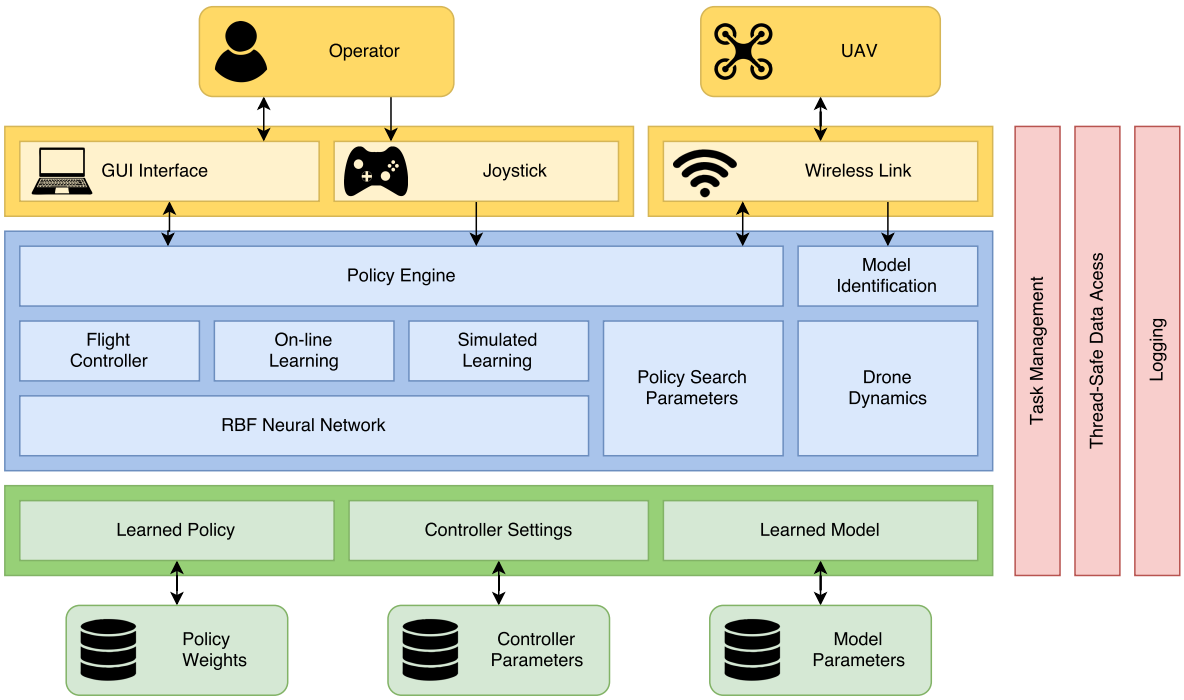


Figure A-1: Software architecture for the intelligent controller

the risks of damage to the hardware. Simultaneously, raw input/output data streaming from the drone is also used for *on-line learning*. This approach allows to update the *flight controller continuously* policy even if a model is not yet available or it is inaccurate, while it also mitigates adverse effects caused by inaccuracies, that might be present in the model, by using real-world data.

The UML diagram of the resulting program is shown in figure A-2. Note that several utility variables and routines had been omitted for the sake of brevity. The core software consists of four principal parts: state management, simulation, the controller itself and the neural net used as a function approximator.

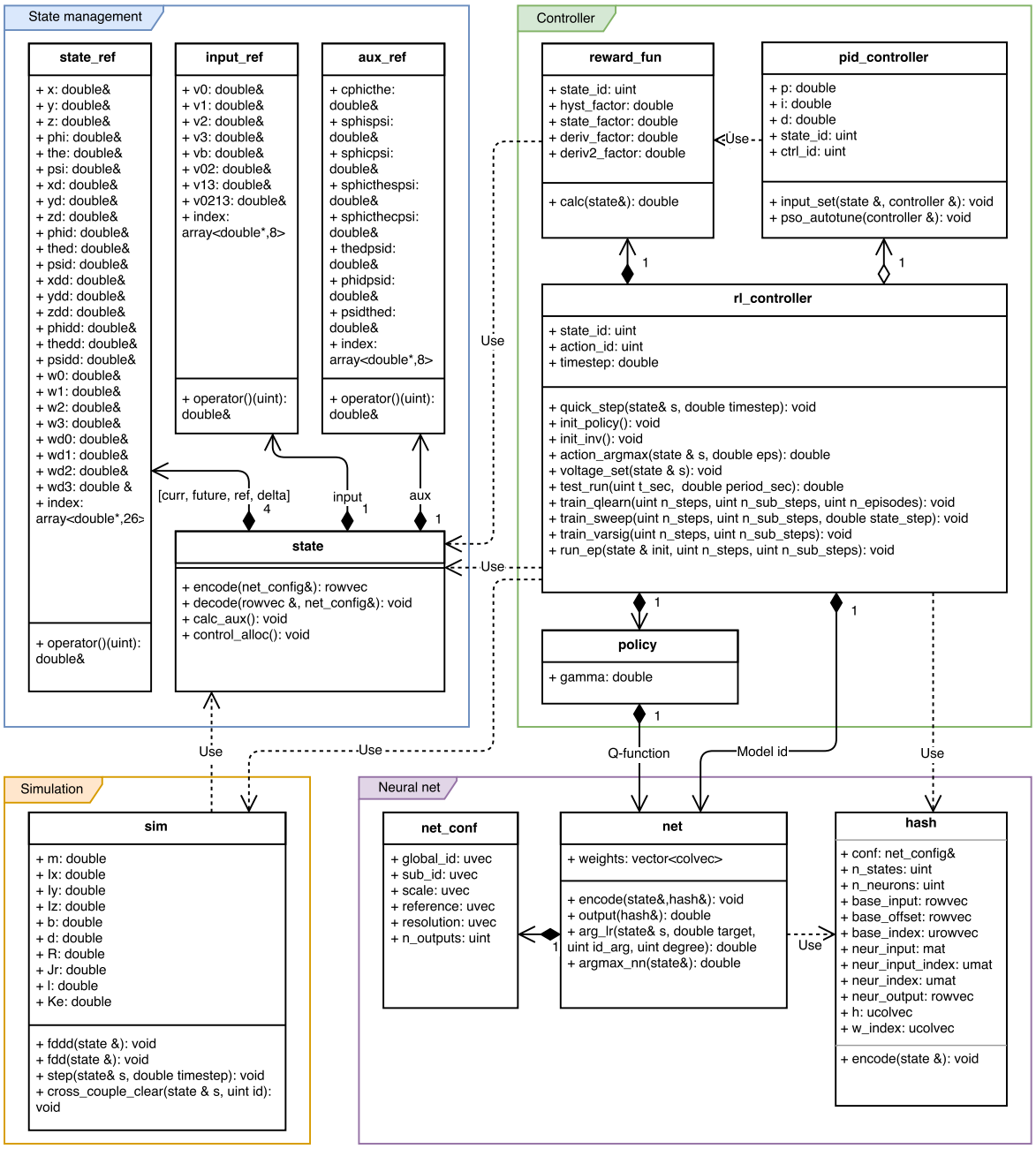


Figure A-2: The experiment software UML diagram

Appendix B

UAV simulation

??

UAV model used has a number of various dynamics that have to be simulated. This section gives a brief overview of the procedure used when creating a computer program used to simulate the drone.

B-0-1 State overview

The model is defined to have a total of 40 states. There are "base" states of the model, such as pitch, roll or yaw angles or their derivatives, the input states and the auxiliary states, defined as a combination of other states. Table B-1 gives an outline of all defined states, implemented as part of the `state` class. The input states are listed in table B-2. Auxiliary states are listed in table B-3. Using reinforcement learning as a mode of control allows addition of any number of cross-coupled states, in order to achieve optimal control. For example adding an additional auxiliary state $\cos(\varphi)\cos(\vartheta)$ in the altitude controller allows it to include the relationship between the aircraft pitch and roll angles and the amount of thrust required to maintain reference altitude, making it an adaptive controller that would apply a different policy, depending on current state of the vehicle.

B-0-2 Dynamics simulation

A numerical solver has to be applied to the model in order to simulate motion of the UAV in flight. There are a number of solver methods available, adaptive Dormand-Prince method for solving of ordinary differential equation was selected(Dormand & Prince, 1980). It is a popular method, used as the default solver algorithm in MATLAB. This method was implemented in the software as part of the `sim` class. The function of a numerical solver is to solve a general differential equation in the form of

$$y' = f(x, t) \tag{B-1}$$

$$y(t_0) = y_0 \tag{B-2}$$

State	Name	SW var.	ID	Min	Max	Unit	Comment
x	X-position	`x`	0	-10.0	10.0	[m]	
y	Y-position	`y`	1	-10.0	10.0	[m]	
z	Altitude	`z`	2	-5.0	5.0	[m]	Altitude controller state
ϕ	Roll-angle	`phi`	3	-180.0	180.0	[deg]	Roll controller state
θ	Pitch-angle	`the`	4	-180.0	180.0	[deg]	Pitch controller state
ψ	Yaw-angle	`psi`	5	-180.0	180.0	[deg]	Yaw controller state
ω_0	Rotor speed (0)	`w0`	6	0.0	530.0	[rad/s]	
ω_1	Rotor speed (1)	`w1`	7	0.0	530.0	[rad/s]	
ω_2	Rotor speed (2)	`w2`	8	0.0	530.0	[rad/s]	
ω_3	Rotor speed (3)	`w3`	9	0.0	530.0	[rad/s]	
$\dot{x}$	X-velocity	`xd`	10	-20.0	20.0	[m/s]	
$\dot{y}$	Y-velocity	`yd`	11	-20.0	20.0	[m/s]	
$\dot{z}$	Rate of climb	`zd`	12	-20.0	20.0	[m/s]	
p	Roll-rate	`phid`	13	-10.0	10.0	[rad/s]	
q	Pitch-rate	`thed`	14	-10.0	10.0	[rad/s]	
r	Yaw-rate	`psid`	15	-2.5	2.5	[rad/s]	
$\dot{\omega}_0$	Rotor accel. (0)	`wd0`	16	-	-	[rad/s^2]	
$\dot{\omega}_1$	Rotor accel. (1)	`wd1`	17	-	-	[rad/s^2]	
$\dot{\omega}_2$	Rotor accel. (2)	`wd2`	18	-	-	[rad/s^2]	
$\dot{\omega}_3$	Rotor accel. (3)	`wd3`	19	-	-	[rad/s^2]	
$\ddot{x}$	X-accel.	`xdd`	20	-10.0	10.0	[m/s^2]	
$\ddot{y}$	Y-accel.	`ydd`	21	-10.0	10.0	[m/s^2]	
$\ddot{z}$	Z-accel.	`zdd`	22	-32.0	12.0	[m/s^2]	Altitude controller action
$\dot{p}$	Roll-accel.	`phidd`	23	-160.0	160.0	[rad/s^2]	Roll controller action
$\dot{q}$	Pitch-accel.	`thedd`	24	-160.0	160.0	[rad/s^2]	Pitch controller action
$\dot{r}$	Yaw-accel.	`psidd`	25	-10.0	10.0	[rad/s^2]	Yaw controller action

Table B-1: Base system states

State	Name	SW var.	ID	Min	Max	Unit	Comment
v_0	Motor voltage (0)	`v0`	26	0.0	12.0	[V]	
v_1	Motor voltage (1)	`v1`	27	0.0	12.0	[V]	
v_2	Motor voltage (2)	`v2`	28	0.0	12.0	[V]	
v_3	Motor voltage (3)	`v3`	29	0.0	12.0	[V]	
v_b	Base differential	`vb`	30	0.0	512.0	[V^2]	$v_b = v_0^2 + v_1^2 + v_2^2 + v_3^2$
v_{02}	Pitch-actuation differential	`v02`	31	-144.0	144.0	[V^2]	$v_{02} = v_0^2 - v_2^2$
v_{13}	Roll-actuation differential	`v13`	32	-144.0	144.0	[V^2]	$v_{13} = v_1^2 - v_3^2$
v_{0213}	Yaw-actuation differential	`v0213`	33	-288.0	288.0	[V^2]	$v_{0213} = v_0^2 - v_1^2 + v_2^2 - v_3^2$

Table B-2: Input states

State	SW var.	ID	Min	Max	Unit	Comment
$c(\phi)c(\theta)$	`cphicthe`	34	-1.0	1.0	[-]	$cos(\phi)cos(\theta)$
qr	`thedpsid`	35	-1.0	1.0	[rad^2/s]	
pr	`phidpsid`	36	-1.0	1.0	[rad^2/s]	
qp	`phidthed`	37	-1.0	1.0	[rad^2/s]	
$c(\phi)s(\theta)c(\psi)$	`cphisthecpsi`	38	-1.0	1.0	[-]	$cos(\phi)sin(\theta)cos(\psi)$
$c(\phi)s(\theta)s(\psi)$	`cphisthespsi`	39	-1.0	1.0	[-]	$cos(\phi)sin(\theta)sin(\psi)$
$s(\phi)s(\psi)$	`sphispsi`	40	-1.0	1.0	[-]	$sin(\phi)sin(\psi)$
$s(\phi)c(\psi)$	`sphicpsi`	41	-1.0	1.0	[-]	$sin(\phi)cos(\psi)$

Table B-3: Auxiliary states

Coefficient	Value	Unit	Description
m	1.2	[kg]	Drone mass
I_x	8.5e-3	[kg·m^2]	Moment of inertia around x-axis
I_y	8.5e-3	[kg·m^2]	Moment of inertia around y-axis
I_z	15.8e-3	[kg·m^2]	Moment of intertia around z-axis
b	2.4e-5	[-]	Thrust factor
d	1.1e-7	[-]	Drag constant
R	2.0	[Ω]	Motor resistance
J_r	1.5e-5	[kg·m^2]	Rotor inertia
J_m	0.5e-5	[kg·m^2]	Motor inertia
l	2.4e-1	[m]	Motor arm length
K_e	1.5e-2	[-]	Motor constant

Table B-4: Drone dynamics coefficients

These equations were described in 6-1 for the translational dynamics, 6-15 for the rotational dynamics, and 6-19 for the motor dynamics. The coefficients used are shown in table B-4.

The one-step calculation of Dormand-Prince solver is done as follows:

$$k_1 = hf(t_k, y_k) \tag{B-3}$$
$$k_2 = hf(t_k + \frac{1}{5}h, y_k + \frac{1}{5}k_1) \tag{B-4}$$
$$k_3 = hf(t_k + \frac{3}{10}h, y_k + \frac{3}{40}k_1 + \frac{9}{40}k_2) \tag{B-5}$$
$$k_4 = hf(t_k + \frac{4}{5}h, y_k + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3) \tag{B-6}$$
$$k_5 = hf(t_k + \frac{8}{9}h, y_k + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2 + \frac{64448}{6561}k_3 - \frac{212}{729}k_4) \tag{B-7}$$
$$k_6 = hf(t_k + h, y_k + \frac{9017}{3168}k_1 - \frac{355}{33}k_2 - \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5) \tag{B-8}$$
$$k_7 = hf(t_k + h, y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6) \tag{B-9}$$

The updated estimate y_{k+1} is then calculated as:

$$y_{k+1} = y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4\frac{2187}{6784}k_5 + \frac{11}{84}k_6 \tag{B-10}$$

This represents a single Runge-Kutta(4) step. In addition to this, another updated estimate is calculated:

$$z_{k+1} = y_k + \frac{5179}{57600}k_1 + \frac{7571}{16695}k_3 + \frac{393}{640}k_4 - \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7 \tag{B-11}$$

This estimate represents a Runge-Kutta(5) step. Combining these two estimates allows to calculate the estimated solver error $|z_{k+1} - y_{k+1}|$

$$|z_{k+1} - y_{k+1}| = |\frac{71}{57600}k_1 - \frac{71}{16695}k_3 + \frac{71}{1920}k_4 - \frac{17253}{339200}k_5 + \frac{22}{525}k_6 - \frac{1}{40}k_7| \tag{B-12}$$

This error estimation can be used to adjust the solver step-size h_{opt}, depending on the acceptable error tolerance value ϵ:

$$s = \left(\frac{\epsilon h}{2|z_{k+1}y_{k+1}|}\right)^{\frac{1}{5}} \tag{B-13}$$

Then the updated optimum step-size becomes

$$h_{opt} = hs \tag{B-14}$$

The application procedure of Dormand-Prince solver is described by algorithm 4.

Data: Initialize $\epsilon = 0.001$, $h = 0.001$, $t_0 = 0$
while $t < t_{fin}$ **do**
  Calculate RK coefficients, using equations of motion described in 6-1, 6-15, and 6-19
  $k_1 = hf(t0, y0)$
  ...
  $k_7 = hf(t0 + h, ...)$
  Calculate updated system state (RK4)
  $y_1 = y_0 + \frac{35}{384}k_1 + \dots$
  Calculate updated system state (RK5) and estimate the resulting error
  $z_1 = y_0 + \frac{5179}{57600}k_1 + \dots$
  $err = |z_1 - y_1|$
  Calculate the optimal step interval for the next step
  $s = \left(\frac{\epsilon h_0}{2err}\right)^{\frac{1}{5}}$
  $h_1 = sh_0$
  Clip the step interval h_1 between h_{min} and h_{max}
  if $h_1 < h_{min}$ **then**
    $h_1 = h_{min}$
  end
  if $h_1 > h_{max}$ **then**
    $h_1 = h_{max}$
  end
  Update the variables
  $t_0 = t_0 + h_0$
  $y0 = y_1$
  $h_0 = h_1$
end
return *The final system state y at* t_{fin}

Algorithm 4: Dormand-Prince solver application

Bibliography

Albus, J. S. (1975). A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, *97*(3), 220. Available from http://dynamicsystems.asmedigitalcollection.asme.org/article.aspx?articleid=1402197

Bagnell, J., & Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, *2*.

Bohlin, T. (1991). *Interactive system identification : Prospects and pitfalls* (Vol. 54) (No. 3). European Journal of Operational Research.

Bohlin, T. (1994). A case study of grey box identification. *Automatica*, *30*(2), 307–318.

Bongard, J., & Lipson, H. (2007). Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America*, *104*, 9943–9948.

Bouabdallah, S., Noth, A., Siegwart, R., & Siegwan, R. (2004). PID vs LQ control techniques applied to an indoor micro quadrotor. *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, *3*, 2451–2456.

Buffington, J., Chandler, P., & Pachter, M. (1999). on-Line System Identification for Aircraft With Distributed Control Effectorsr. *International Journal of Robust and Nonlinear Control*, *1049*, 1033–1049.

Buffington, J. M., & Enns, D. F. (1996, nov). Lyapunov stability analysis of daisy chain control allocation. *Journal of Guidance, Control, and Dynamics*, *19*(6), 1226–1230. Available from http://arc.aiaa.org/doi/abs/10.2514/3.21776%5Cnhttp://arc.aiaa.org/doi/10.2514/3.2177 http://arc.aiaa.org/doi/10.2514/3.21776

Busoniu, L., Babuška, R., De Schutter, B., & Ernst, D. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators* (1st ed.). Boca Raton, FL, USA: CRC Press, Inc.

Carroll, J., Peterson, T., & Owens, N. (2001). Memory-guided exploration in reinforcement learning. *IJCNN'01. International Joint Conference on Neural Net-*

works. Proceedings (Cat. No.01CH37222), *2*(January), 1–44. Available from http://portal.acm.org/citation.cfm?id=865072

Cohn, D., Atlas, L., & Ladner, R. (1994). Improving generalization with active learning. *Machine Learning*, *15*(2), 201–221.

Cox, C., Stepniewski, S., Jorgensen, C., Saeks, R., & Lewis, C. (1999). On the design of a neural network autolander. *International Journal of Robust and Nonlinear Control*, *9*(14), 1071–1096.

Cybenko, G. (1998). Neuro-Dynamic Programming. *IEEE Computational Science and Engineering*, *5*(2), 101–102. Available from http://www.amazon.com/dp/1886529108 http://ieeexplore.ieee.org/document/683749/

Dormand, J. R., & Prince, P. J. (1980). A family of embedded Runge- Kutta formulae. *Journal of computational and applied mathematics*, *6*(1), 19–26. Available from http://www.sciencedirect.com/science/article/pii/0771050X80900133

Durham, W. C. (1994, mar). Constrained control allocation - Three-moment problem. *Journal of Guidance, Control, and Dynamics*, *17*(2), 330–336. Available from http://arc.aiaa.org/doi/10.2514/3.21201

Gaing, Z.-L. L. (2004). A Particle Swarm Optimization Approach for Optimum Design of PID Controller in AVR System. *IEEE Transactions on Energy Conversion*, *19*(2), 384–391. Available from http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1300705&url=http://ieeexplore.i

Gaskett, C. (2008). Q-learning for robot control. *Dspace.Anu.Edu.Au*, *1*. Available from http://dspace.anu.edu.au/handle/1885/47080%5Cnpapers2://publication/uuid/EE6874D9-54E

Gaskett, C., Wettergreen, D., & Zelinsky, A. (1999). Q-learning in continuous state and action spaces. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 1747, pp. 417–428). Available from http://www.springerlink.com/content/2178756r7k338683

Geibel, P., & Wysotzki, F. (2005). Risk-sensitive reinforcement learning applied to control under constraints. *Journal of Artificial Intelligence Research*, *24*, 81–108.

Geramifard, A., Redding, J., & How, J. P. (2013). Intelligent Cooperative Control Architecture: A Framework for Performance Improvement Using Safe Learning. *Journal of Intelligent & Robotic Systems*, *72*(1), 83–103. Available from http://link.springer.com/10.1007/s10846-013-9826-6

Hageman, J. J., Smith, M. S., & Stachowiak, S. (2003). Integration of Online Parameter Identification and Neural Network for In-Flight Adaptive Control. *Report Project-NASA*(October).

Harkegard, O. (2002). Efficient active set algorithms for solving constrained least squares problems in aircraft control allocation. *Decision and Control, 2002, Proceedings of the ...*, *2*(December), 1295–1300. Available from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1184694

Harmon, M., & Baird III, L. (1996). Residual advantage learning applied to a differential game. *Neural Networks, 1996., IEEE International Conference on*, *1*(June), 329–334 vol.1.

Hassan, R., & Cohanim, B. (2005). A comparison of particle swarm optimization and the genetic algorithm. *1st AIAA multidisciplinary design optimization specialist conference*, 1–13. Available from http://arc.aiaa.org/doi/pdf/10.2514/6.2005-1897

Hasselt, H. V., Group, A. C., & Wiskunde, C. (2010). Double Q-learning. In *Neural information proceeding systems* (pp. 1–9).

Hillier, F. S., Lieberman, G. J., & Newton, S. I. (2015). Introduction to operations research. In (pp. 1–14). New York, NY: McGraw-Hill. Available from http://opac.inria.fr/record=b1082814

Iruthayarajan, M. W., & Baskar, S. (2009). Evolutionary algorithms based design of multivariable PID controller. *Expert Systems with Applications*, *36*(5), 9159–9167. Available from http://dx.doi.org/10.1016/j.eswa.2008.12.033

Jorgensen, C. C. (1997). Direct Adaptive Aircraft Control Using Dynamic Cell Structure Neural Networks. *Cell Structure Neural Networks", NASA TM 112198*(May).

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning : A Survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kurnaz, S., Cetin, O., & Kaynak, O. (2010). Adaptive neuro-fuzzy inference system based autonomous flight control of unmanned air vehicles. *Expert Systems with Applications*, *37*(2), 1229–1234. Available from http://dx.doi.org/10.1016/j.eswa.2009.06.009

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

Mohammadi, M., Shahri, A. M., & Boroujeni, Z. (2012, jan). Modeling and Adaptive Tracking Control of a Quadrotor UAV. *International Journal of Intelligent Mechatronics and Robotics*, *2*(4), 58–81. Available from http://www.google.com/url?sa=t&rct=j&q=trajectory tracking control of quadrotor uav&source=web&cd=6&ved=0CGoQFjAF&url=http://www.atlantis-press.com/php/dow

Mukherjee, V., & Ghoshal, S. P. (2007). Comparison of intelligent fuzzy based AGC coordinated PID controlled and PSS controlled AVR system. *International Journal of Electrical Power and Energy Systems*, *29*, 679–689.

Narendra, K. S., & Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, *1*(1), 4–27.

Neumann, G. (2005). *The Reinforcement Learning Toolbox , Reinforcement Learning for Optimal Control Tasks*. Unpublished doctoral dissertation.

Nievergelt, J. (1969). R69-13 Perceptrons: An Introduction to Computational Geometry. *IEEE Transactions on Computers*, *C-18*(6), 572–572. Available from http://cdsweb.cern.ch/record/114106 http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1671311

Pitts, W., & McCulloch, W. S. (1947). How we know universals the perception of auditory and visual forms. *The Bulletin of Mathematical Biophysics*, *9*, 127–147.

Poli, R. (2007). An Analysis of Publications on Particle Swarm Optimisation Applications. *Journal of Artificial Evolution and Applications*, *2008*, 1–57.

Poli, R. (2008). Analysis of the Publications on the Applications of Particle Swarm Optimisation. *Journal of Artificial Evolution and Applications*, *2008*(2), 1–10.

Simon, S. L., Duncan, C. L., Horky, S. C., Nick, T. G., Castro, M. M., & Riekert, K. a. (2011). Body satisfaction, nutritional adherence, and quality of life in youth with cystic fibrosis. *Pediatric Pulmonology*, *46*(11), 1085–1092. Available from http://www.ncbi.nlm.nih.gov/pubmed/21626713

Sjöberg, J., Zhang, Q., Ljung, L., Benveniste, A., Delyon, B., Glorennec, P.-Y., et al. (1995).

Nonlinear black-box modeling in system identification: a unified overview (Vol. 31) (No. 12).

Sutton, R. S., & Barto, A. G. (1998). Reinforcement learning: an introduction. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, *9*, 1054.

Svendsen, C. H., Holck, N. O., Galeazzi, R., & Blanke, M. (2012). L1 adaptive manoeuvring control of unmanned high-speed water craft. In *Proceedings of the 9th ifac conference on manoeuvring and control of marine crafts* (Vol. 9, pp. 144–151). Arenzano: International Federation of Automatic Control.

Valasek, J., Doebbler, J., Tandale, M. D., & Meade, A. J. (2008). Improved adaptive-reinforcement learning control for morphing unmanned air vehicles. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *38*(4), 1014–1020.

Valasek, J., Tandale, M. D., & Rong, J. (2005). A Reinforcement Learning - Adaptive Control Architecture for Morphing. *Journal of Aerospace Computing, Information, and Communication*, *2*(April), 174–195.

Whitehead, S. D. (1991). A Complexity Analysis of Cooperative Mechanisms in Reinforcement Learning. *AAAI-91 Proceedings*, 607–613.

Zadeh, L. (1975). The concept of a linguistic variable and its application to approximate reasoning—II. *Information Sciences*, *8*(4), 301–357.

Zadeh, L. a. (1973). Outline of a New Approach to the Analysis of Complex Systems and Decision Processes. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-3*(1), 28–44.

Zhang, B., Mao, Z., Liu, W., & Liu, J. (2013). Geometric Reinforcement Learning for Path Planning of UAVs. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 1–19.