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

Naruta, Anton; Mannucci, Tommaso; van Kampen, Erik-jan

DOI
10.2514/6.2019-0145

Publication date
2019

Document Version
Accepted author manuscript

Published in
AIAA Scitech 2019 Forum

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

Anton E. Naruta*, Tommaso Mannucci†, Erik-Jan van Kampen‡

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.[5–13]

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,

*MSc student, Control and Simulation Division, Faculty of Aerospace Engineering, Delft University of Technology
†PhD candidate, Control and Simulation Division, Faculty of Aerospace Engineering, Delft University of Technology
‡Assistant Professor, Control and Simulation Division, Faculty of Aerospace Engineering, Delft University of Technology

and exceeds it. Additionally, it is demonstrated that the RL controller is capable of adjusting its policy online if agent behavior deviates from the initial model used to train the policy.

II. Reinforcement learning

Reinforcement learning and intelligent control are used increasingly in various control applications. Reinforcement learning can deal with the non-linearities 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 deemed optimal in the situation.[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 behaviour B. The latter 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 modelled 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 change of state, 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 behaviour, 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.[14] 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.[15] 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*.[16,17] 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 summation 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:

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

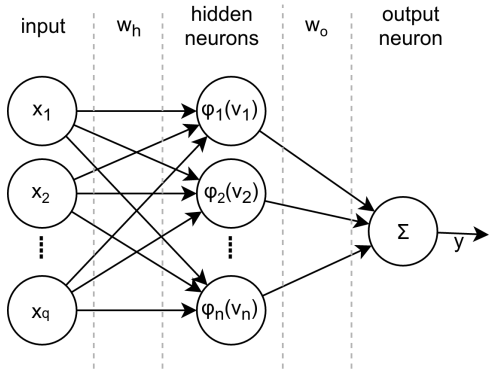


Figure 1. Neural net layout

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,[18] 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 .[19] 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 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, the total number of neurons in the network has no effect on its performance. 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:

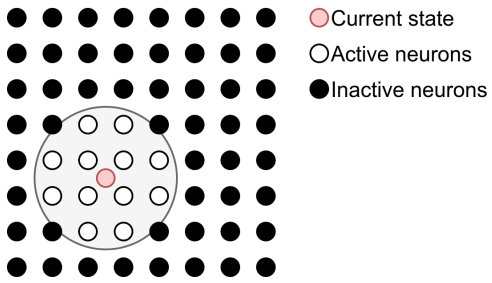


Figure 2. Neuron activation scheme

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

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 %)

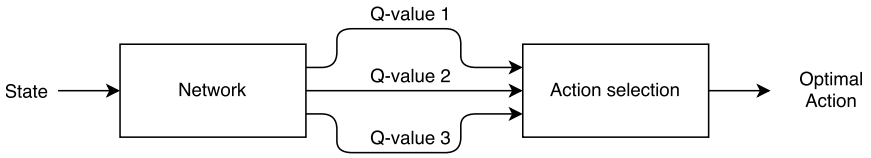
Table 1. Number of activated neurons for selected dimensions

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.

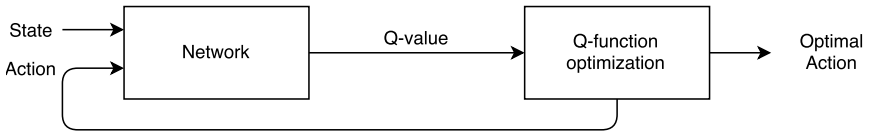
D. Optimal action selection

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



(a) Discrete action



(b) Continuous action

Figure 3. Q-learning action selection approaches

method typically requires several passes. A limited number of forward passes means it's less demanding 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

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

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 $\dot{y}(a)$:

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

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 modelling 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 of 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.[20,21] 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.[20] The motor dynamics are described in equation 39.

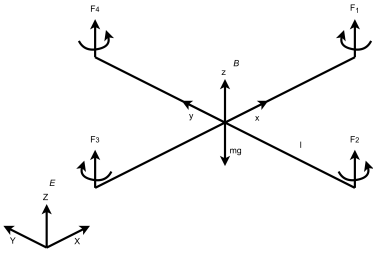


Figure 4. Quadrotorotor configuration schematic, body fixed frame B and inertial frame E.

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

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

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

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

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

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

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

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

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

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

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

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

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 behaviour of the model.

The full drone controller consists of 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 behaviour. Higher-level outer loop controller is designed to track a position reference, such as x and y coordinates of the vehicle, and perform manoeuvres.

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 manoeuvres, 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.

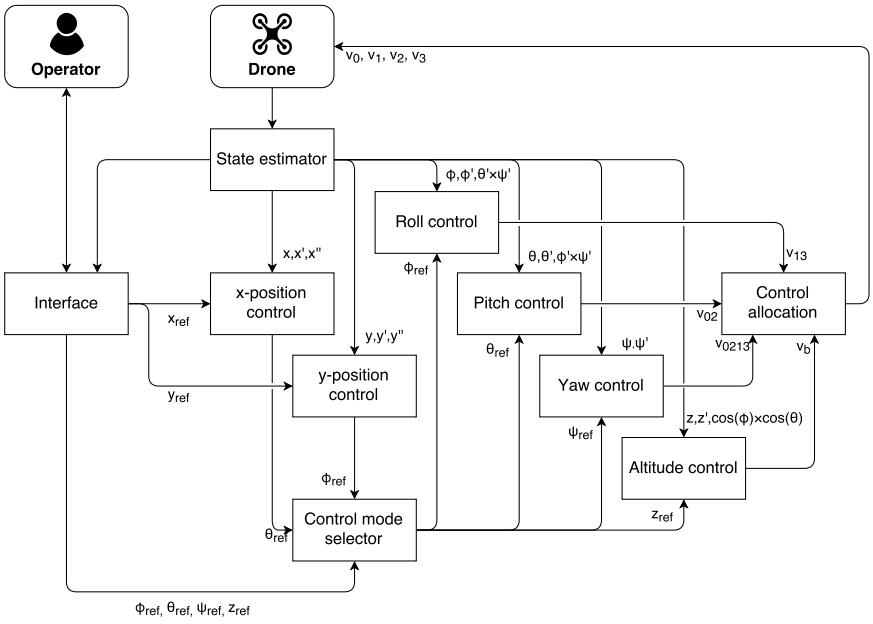


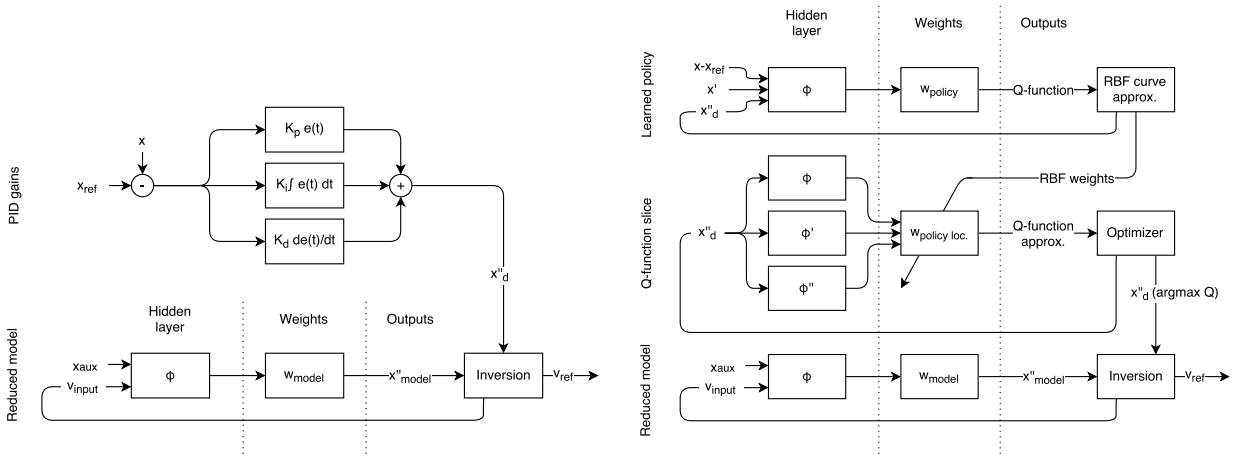
Figure 5. Drone control scheme

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.[22–24] 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 behaviour.

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.

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 manoeuvres. More aggressive manoeuvring might result in a larger deviation.

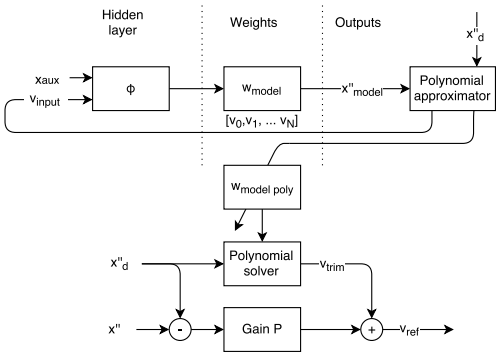
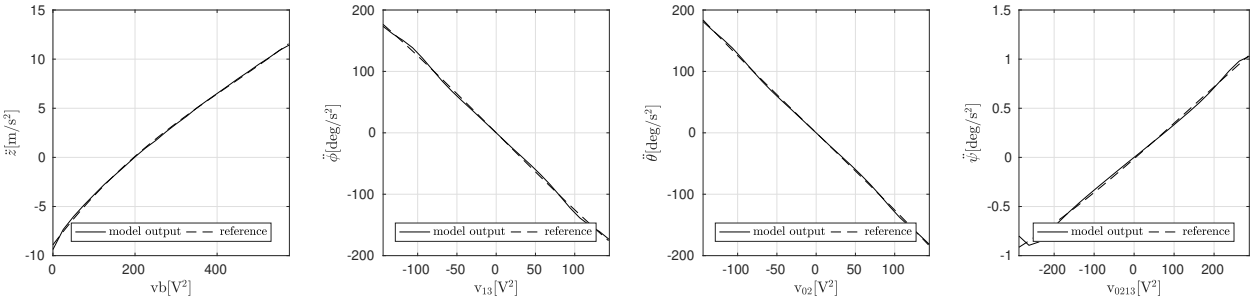


Figure 7. Inversion controller

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

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

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

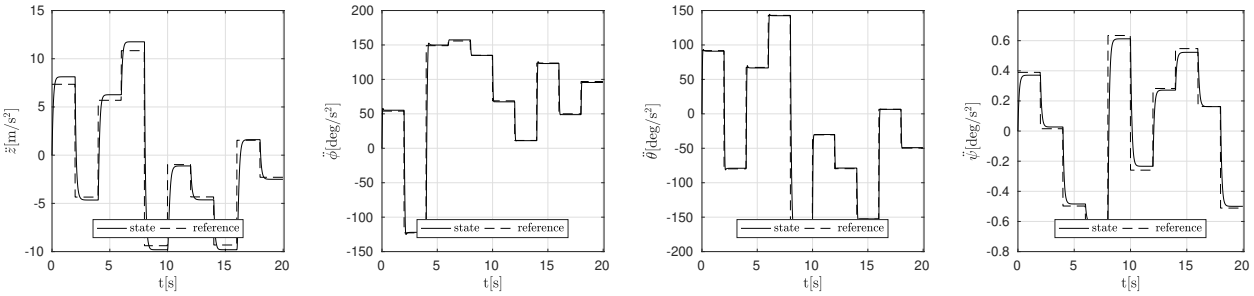
$$\theta^* = \begin{bmatrix} p_0 + \ddot{x}_d & p_1 & \ldots & p_N \end{bmatrix}^T. \tag{42}$$

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 a terminal reward is assigned. 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 behaviour 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[25] 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,[26] 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 the steady-state performance of the controller, while the sinusoidal signal is designed to evaluate the response of the controller to a moving reference signal.

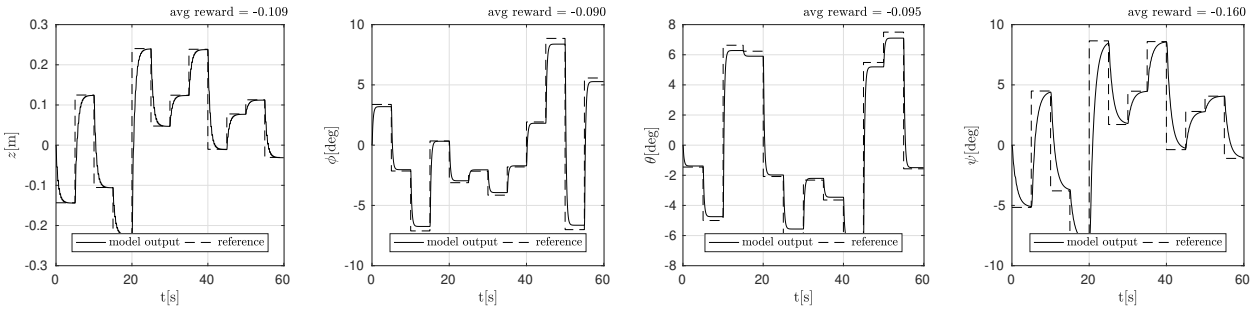
There are three training methods used to train the policy: random state sweep, grid-based state sweep and variable signal response. In a random state sweep, the initial state is placed randomly, close to the final goal state. As the learning progresses, the initial state is placed further and further 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

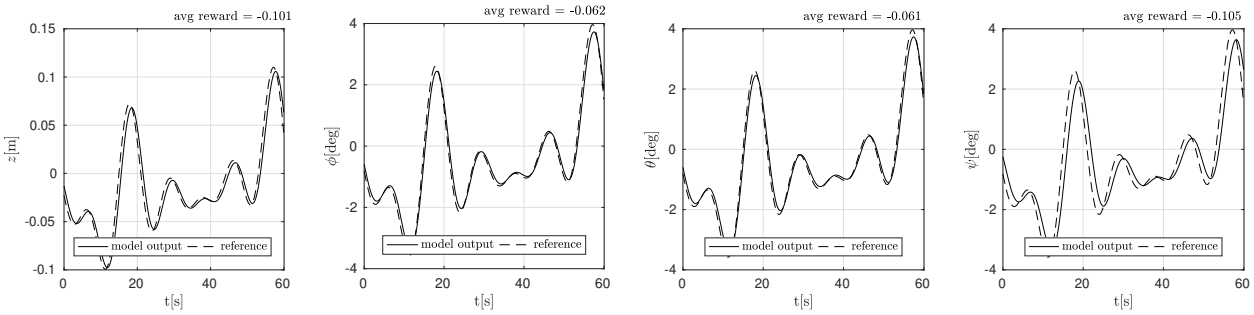
To evaluate the performance of the trained system and compare it against the PD two types of signals are used: a variable step input and a sinusoidal pseudo-random input. The controllers are evaluated in isolation - e.g., only one controller is activated at the time for each dynamic mode.

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

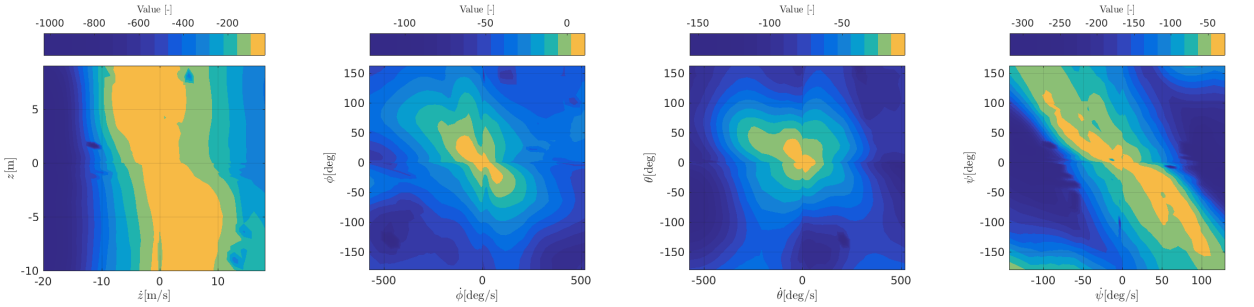


(a) Vertical motion controller (b) Roll controller (c) Pitch controller (d) Yaw controller

Figure 11. PID controller sine wave signal response

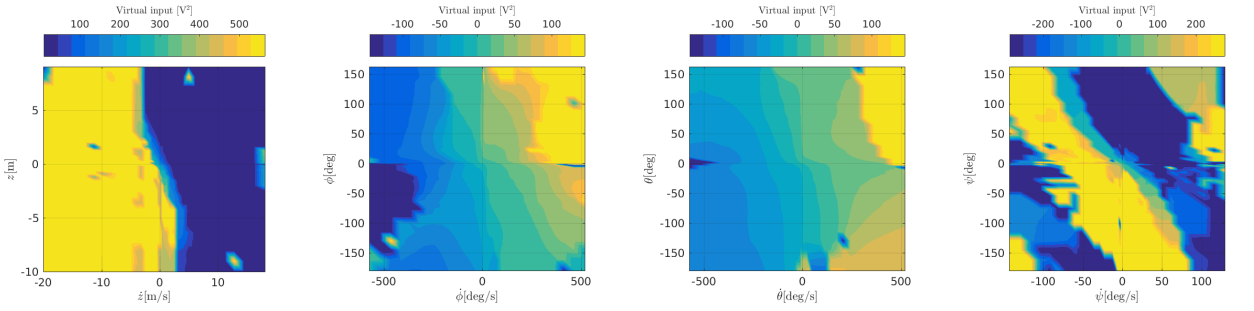
The PID controller performance is scored by calculating the average of all rewards received through the test trial. During these trials the full model of the process is used(with motor dynamics included). The resulting score allows to make an objective comparison against the performance of a reinforcement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



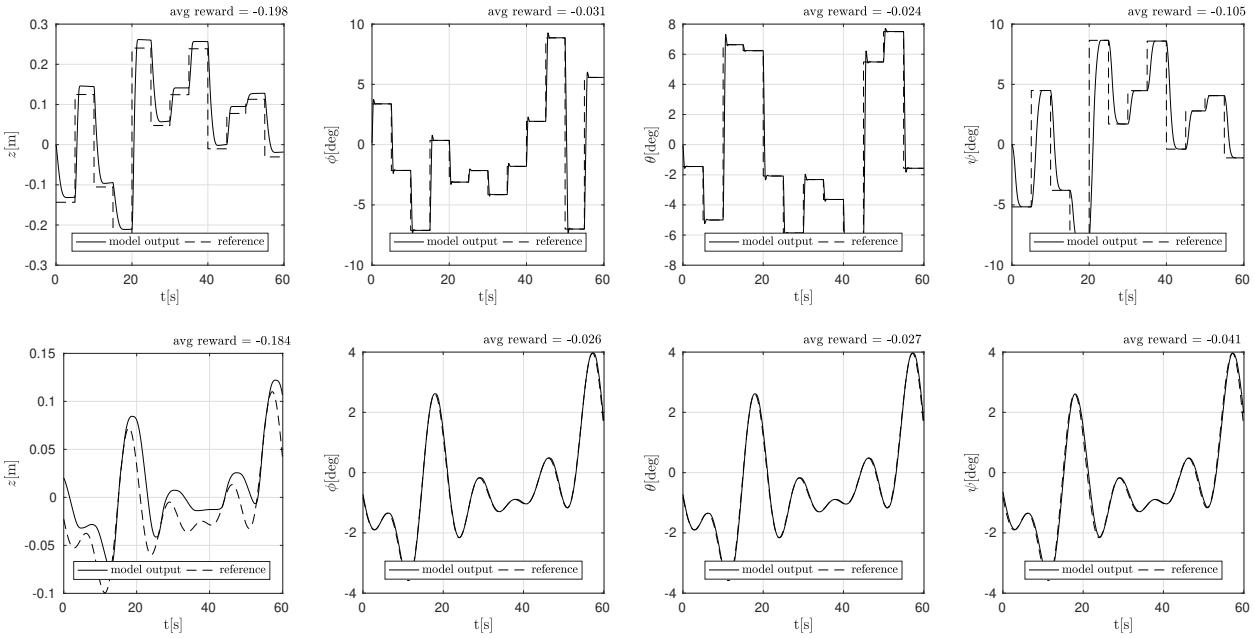
(a) Altitude controller (b) Roll controller (c) Pitch controller (d) Yaw controller

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.

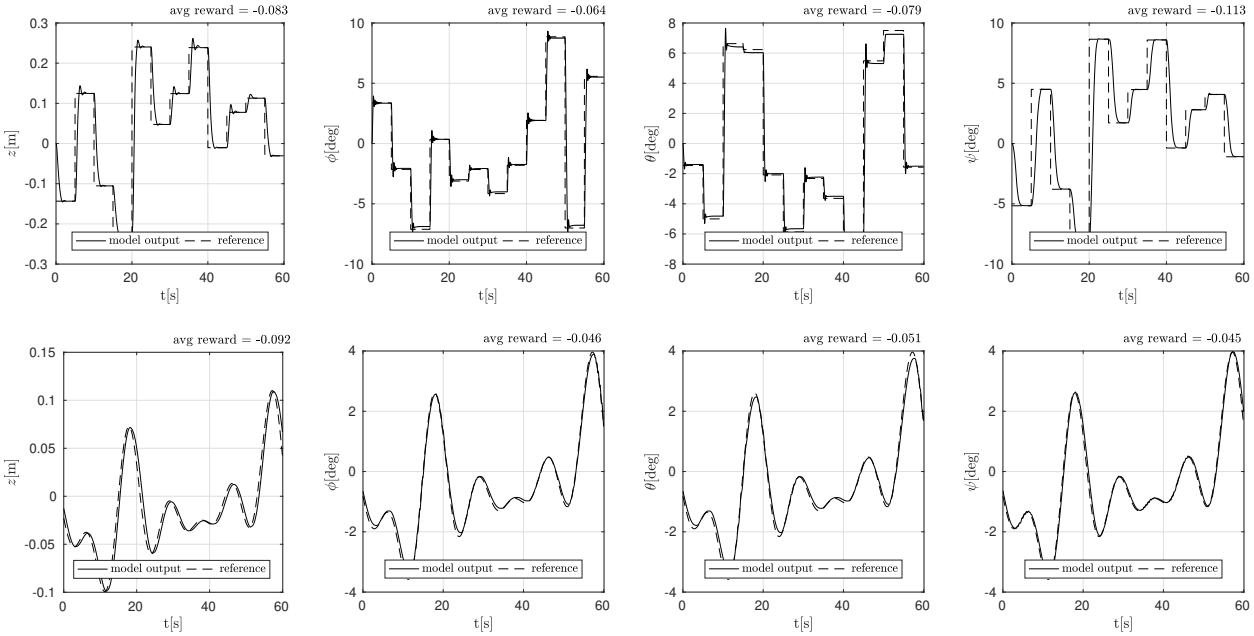
Another interesting feature is the behaviour of the Yaw controller. The action map looks like a sequence of bands. This nonlinearity can be explained by the fact that the yaw controller has the least actuation power, reacting very slowly and the states themselves are cyclic, meaning that as the aircraft 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 the reduced model in most cases. A complete list of test run scores is 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 the inner-loop 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 an 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]van Kampen, E.-J., Chu, Q., and Mulder, J., "Continuous adaptive critic flight control aided with approximated plant dynamics," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2006, p. 6429.

[10]Zhou, Y., Kampen, E.-J. v., and Chu, Q., "Nonlinear adaptive flight control using incremental approximate dynamic programming and output feedback," *Journal of Guidance, Control, and Dynamics*, Vol. 40, No. 2, 2016, pp. 493–496.

[11]Zhou, Y., Van Kampen, E., and Chu, Q. P., "Incremental approximate dynamic programming for nonlinear flight control design," *Proceedings of the 3rd CEAS EuroGNC: Specialist Conference on Guidance, Navigation and Control, Toulouse, France, 13-15 April 2015*, 2015.

[12]Zhou, Y., van Kampen, E., and Chu, Q., "Incremental model based heuristic dynamic programming for nonlinear adaptive flight control," *Proceedings of the International Micro Air Vehicles Conference and Competition 2016, Beijing, China*, 2016.

[13]Mannucci, T., van Kampen, E.-J., de Visser, C., and Chu, Q., "Safe exploration algorithms for reinforcement learning controllers," *IEEE transactions on neural networks and learning systems*, Vol. 29, No. 4, 2018, pp. 1069–1081.

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

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

[16]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.

[17]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.

[18]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.

[19]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.

[20]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.

[21]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.

[22]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.

[23]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.

[24]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.

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

[26]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.