Flexible Heuristic Dynamic Programming for Reinforcement Learning in Quadrotors

Helmer, Alexander; de Visser, Coen; van Kampen, Erik-Jan

DOI

[10.2514/6.2018-2134](https://doi.org/10.2514/6.2018-2134)

Publication date

2018

Document Version

Accepted author manuscript

Published in

Proceedings of the 2018 AIAA Information Systems-AIAA Infotech @ Aerospace

Citation (APA)

Helmer, A., de Visser, C., & van Kampen, E.-J. (2018). Flexible Heuristic Dynamic Programming for Reinforcement Learning in Quadrotors. In *Proceedings of the 2018 AIAA Information Systems-AIAA Infotech @ Aerospace* [AIAA 2018-2134] American Institute of Aeronautics and Astronautics Inc. (AIAA). <https://doi.org/10.2514/6.2018-2134>

Important note

To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright

Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy

Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

Flexible Heuristic Dynamic Programming for Reinforcement Learning in Quad-Rotors

A.M.C. Helmer*, C.C. de Visser†, and E. van Kampen‡

Delft University of Technology, Delft, 2600 GB, The Netherlands

Reinforcement learning is a paradigm for learning decision-making tasks from interaction with the environment. Function approximators solve a part of the curse of dimensionality when learning in high-dimensional state and/or action spaces. It can be a time-consuming process to learn a good policy in a high dimensional state space directly. A method is proposed for initially limiting the state and action space to a subset of the variables of the Markov Decision Process. Therefore, the agent will initially learn a coarse policy. It is then gradually exposed to new state and action variables to increase the dimensionality of the state and action space to the ones posed by the control problem. A local function approximator has been developed that supports the expansion of state and action space. The concept is applied to the Model-Learning Actor-Critic, a model-based Heuristic Dynamic Programming algorithm. Its functioning is demonstrated by training a reinforcement learning agent for 2-dimensional hover control of a Parrot AR 2.0 quad-rotor. It is shown that the agent is able to learn faster and to achieve a better policy when being exposed to the action and state variables gradually than all at once from the start.

Nomenclature

α	Learning rate
δ	Temporal-difference error
ϵ	Tolerance for deciding whether to add a sample to the memory
γ	Discount factor for future rewards
$\hat{\beta}$	Estimated parameters
$\hat{f}_{\xi}$	Plant model, parametrized by ξ
κ	Scale parameter for rotor RPM
λ	Eligibility trace factor
μ	Mean
ω	Rotor speed [RPM]
π_{η}	Policy, parametrized by η
ρ	Reward function
σ	Standard deviation
θ	Pitch angle
A	Action space
a	Action
b	Body reference system
D	Distribution of initial states
e	State error
F	Force due to rotor velocity [N]
i	Inertial reference frame
j	Interval for exploration

*MSc Student, Control and Simulation Division, Faculty of Aerospace Engineering, a.m.c.helmer@student.tudelft.nl

†Assistant Professor, Control and Simulation Division, Faculty of Aerospace Engineering, c.c.devisser@tudelft.nl

‡Assistant Professor, Control and Simulation Division, Faculty of Aerospace Engineering, e.vankampen@tudelft.nl

k	Number of nearest neighbors
M	Memory, and moment due to rotor velocity [Nm]
N	Memory size
P	Transition distribution function
p	Sample in memory
R	Sum of discounted future rewards
r	Reward
S	State space
s	System state
T	Episode length [s]
t	Time [s]
v	Value according to value function V
V_ζ	Value function, parametrized by ζ
W	Weight matrix for quadratic reward function
w	Scale factor for variable in KD-Tree
X	Regression matrix
Y	Target matrix for regression
z	Eligibility trace

I. Introduction

Reinforcement Learning (RL) is a method for learning decision-making tasks from interaction with the environment.¹ It is based on one of the ways in which humans and animals learn; by taking exploratory actions and evaluating the result of their action. Good actions are reinforced by numerical rewards, so the agent (controller) will be more inclined to take that action in the future. By repeatedly moving around in the environment, revisiting states and exploring new actions, the agent will learn which states and actions are favorable as they yield the higher total rewards than others.

When learning a new control task, humans develop an on-board model of the dynamics of the environment, e.g. the response of a car while learning to drive. To speed up learning in early stages we tend to constrain the model to a limited set of inputs and outputs. One may start by only slightly tapping the gas pedal to see how the car responds by moving forward, not taking into account shifting, turning or braking. When a coarse feeling has been developed of how the car moves forward, the state space can be expanded to other state variables and inputs, such as heading and lateral control through the steering wheel. In humans, this speeds up learning and improves the safety of exploration during learning.

This article proposes a method to extend this idea to model-based RL. A flexible Heuristic Dynamic Programming (HDP) algorithm is developed for continuous state/action space RL that enables faster learning by first having the agent learn a coarse policy using subspaces of the state and action spaces. The policy is learned by using only a subset of all state and action variables of the Markov Decision Process (MDP). New variables are added gradually during which the agent employs prior experience as a basis to further build experience. To achieve this, a function approximator (FA) has been developed that is flexible in its structure, and can gain dependency on new variables, as more variables will become available during training. These function approximators will be referred to as *flexible function approximators* (FFAs), as opposed to their non-flexible counterpart, the *static function approximators* (SFAs). The FFA developed in this article is an adaptation of a Local Linear Regression (LLR) non-parametric FA.

In a way, this relates to transfer learning and continual learning. Transfer learning concerns the transfer of experience learned during prior tasks to new tasks.²³⁴⁵ Humans are able to generalize very well between activities, but machines often lack this skill. Continual learning is similar to transfer learning in the sense that it attempts to use prior experience to solve new tasks. As opposed to transfer learning, these tasks have to be increasingly complex. An example is the CHILd architecture,⁶ which is able to learn how to navigate in increasingly complex grid-world mazes. The approach in this article leverages the transfer of experience not between tasks, but rather for the same task with an increasing degree of freedom.

The method in this article is an adaptation of the Model-Learning Actor-Critic (MLAC).⁷ MLAC is a model-based actor-critic method that employs a policy gradient method to update its actor. The flexible MLAC is tested in the RL task of 2D hover control of a quad-rotor. Quad-rotor control poses a challenging test case, due to its unstable nature, non-linear behavior and state space dimensionality.

Section II introduces RL and model-based RL. The policy gradient algorithm and FAs are shown in section III. Section IV discusses the concept of FFAs. The experiment setup and results are discussed in section V and section VI respectively. Finally in section VII, the main conclusions are highlighted and recommendations for future research are given.

II. Reinforcement learning

In RL, an agent learns to control a process by interacting with it. It chooses an action a_t when in state s_t , causing it to transition to state s_{t+1} and receiving reward r_{t+1} , as shown in Figure 1. By iteratively traversing the environment, the agent is able to learn which states and actions yield it the highest cumulative reward and are therefore desirable.

Figure 1. Schematic of the interaction between an agent and its environment.¹

II.A. The Markov Decision Process

The formalization of many RL problems is given by the Markov Decision Process (MDP).¹ In RL, the agent has to learn to make optimal decisions a_t^* based on state information s_t . In particular cases, only the state of the system at time t suffices to choose an action, meaning no other information, such as previous states, previous actions or previous rewards, are required for the decision making process. Such a process is said to have the *Markov* property, which can be formalized as follows.

Consider a process with a sequence of past states $s_t, s_{t-1}, \dots, s_0$, actions $a_t, a_{t-1}, \dots, a_0$ and rewards $r_t, r_{t-1}, \dots, r_1$. The state signal is said to have the Markov property if and only if the probability of transitioning to state s' at $t + 1$ depends *only* on parameters s and a at time t and can hence be expressed as in Equation 1.¹

$$\Pr \{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} \quad (1)$$

An MDP can be represented as a tuple $\langle S, D, A, P(s_t, a_t), \gamma, \rho(s_t, a_t) \rangle$, where S is the set of possible states of the system (the state space), D the distribution of initial states over S , A the set of possible actions of the system, $P(s_t, a_t)$, the state transition distribution of transitioning to state s' when in state s_t and choosing action a_t , $\gamma \in [0, 1)$ the discounting factor, and $\rho(s_t, a_t)$, the reward function that yields instantaneous reward r for encouraging and discouraging particular states and actions.

II.B. Model-based Reinforcement Learning

In model-based RL the agent learns a model of the dynamics of the environment. The so-called *plant model* is an approximation of the state transition distribution $P(s_t, a_t)$, which for a given state and action returns the estimated next state $\hat{s}' = \hat{f}(s_t, a_t)$, or a distribution of potential next states in case of a stochastic environment.⁸ It is particularly useful in training as it can be employed for planning future actions, rolling out trajectories or computing mathematical properties of the environment such as the derivatives of future states with respect to the action. Conversely, algorithms that do not exhibit this behavior are classified as model-free methods.

Well-known examples of model-based RL algorithms are experience replay,⁹ prioritized sweeping,¹⁰ and Dyna,¹¹ which all employ a discretized plant model. Using a continuous FA for the plant model introduces another way to improve training: policy gradients. As the model provides a mathematical description of the plant, it can be used to determine the gradient of a future state with respect to the actions. Together with a

value function, the derivative of the value with respect to the action can be computed. Evidently, the value function needs to be continuous and differentiable to achieve this.

Model-based and model-free methods both have their advantages and disadvantages. On the positive side, model-based methods can employ their plant model for mental planning and for additional training (e.g. through experience replay). Furthermore, plant models may enhance the safety of training, when training physical systems on-line. Model-based learning systems also allow for expert knowledge to be included in the plant model architecture.¹² On the other hand, particular model types require significant computational resources. Furthermore, the integration of a plant model in the RL algorithm adds extra complexity to the system. Models may lag upon plant changes, such that new behavior may not be incorporated in the model immediately (if at all) and the agent may keep on training on invalid, outdated dynamics. Finally, models that are entirely trained on-line may have poor performance at the start of training, effectively slowing down the training of the controller due to false cues.

II.C. Taxonomy of RL algorithms

Many algorithms exist that can solve MDPs. Three main categories can be distinguished; dynamic programming, Monte-Carlo methods and Temporal-Difference (TD) methods.¹ As dynamic programming requires perfect knowledge of the transition distribution of the system $P(s_t, a_t)$, and a known state space and action space, it is rarely used. Monte-Carlo methods are unable to compute the exact value of a state, but rather approximate it by rolling out trajectories. For continuous state/action spaces this is often computationally intractable. The last category, TD methods, can be subdivided in actors, critics and actor-critics. Critics (e.g. Q-Learning¹³ and SARSA^{14,15}) learn a value function and pick the action which leads them to the next state with the highest value. As they enumerate over the action space, critics can only be employed for discrete action spaces. Actors (e.g. Stochastic Real-Valued algorithm¹⁶ and the REINFORCE algorithm¹⁷) learn an explicit policy and optimize directly over the parameter space of the policy by means of a cost or cost-to-go function. Actor-critics learn a separate value function and policy.

Actor-critic methods can be subdivided in three categories;¹⁸ Heuristic Dynamic Programming (HDP) can be seen as the default actor-critic method, where the critic approximates the sum of discounted future rewards, also known as the value function. In Dual Heuristic Programming (DHP) the critic does not estimate the value function directly, but rather its gradient with respect to the state variables. Lastly, in Global Dual Heuristic Programming (GDHP) both the value function as well as its gradient are computed. All three methods also have an Action-Dependent (AD-) version, where not only the state, but also the action is supplied to the critic as an input.

III. The model-learning actor-critic

MLAC, an HDP model-based actor-critic method, is used as a basis for the flexible HDP algorithm.⁷ It has been chosen because it is one of the few continuous model-based RL algorithms that learns the plant model on-line. This is a requirement for FFAs when using policy gradient methods, as information about new state and action variables needs to be learned on-line.

MLAC is a model-based extension of the Standard Actor-Critic (SAC) method¹⁹ that employs a policy gradient method to learn. Similar to SAC, both methods contain a separate actor and critic and approximate them using two separate FAs. The policy provided by the actor is defined as $a_t = \pi_\eta(s_t)$, parametrized by η , and the value function provided by the critic is defined as $V_\zeta^\pi(s_t) \approx R^\pi(s_t)$, with ζ the parametrization of the value function and $R^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$ the sum of future discounted rewards. As opposed to SAC, MLAC also learns an approximate plant model $\hat{s}' = \hat{f}_\xi(s_t, a_t)$, parametrized by ξ , on-line. The optimal action a^* in state s can be found by solving

$$a^* = \pi^*(s_t) = \arg \max_{a' \in A} [\rho(s_t, a', \hat{f}_\xi(s_t, a')) + \gamma V(\hat{f}_\xi(s_t, a'))]. \quad (2)$$

However, when dealing with continuous action spaces, it is impossible to enumerate over all actions to find $\arg \max_{a' \in A}$. Hence the plant model is used to aid in computing the policy gradient by providing the derivative of the next state with respect to the action. In a continuous action space, Equation 2 can be optimized by incrementing the policy parameters η in the direction of the maximum sum of future discounted

rewards using gradient ascent, as shown in Equation 3.

$$\eta \leftarrow \eta + \alpha_{ac} \nabla_{\eta} R^{\pi}(s_t), \quad (3)$$

with α_{ac} the learning rate for the actor.

The partial derivative of the value function with respect to the action $\nabla_{\eta} R^{\pi}(s_t)$ is not readily available, but can be estimated by splitting up the derivative using the chain rule, as shown in Equation 4. This again shows the purpose of learning a plant model, which is to provide $\frac{\partial \hat{s}_{t+1}}{\partial a_t}$. The other partial derivatives are computed from the reward function, the critic and the actor.

$$\nabla_{\eta} R(s_t) \approx \nabla_{\eta} V_{\zeta}(s_t) = \left\{ \frac{\partial \rho(s_t, a_t)}{\partial a_t}^{\top} + \gamma \frac{\partial V_{\zeta}(\hat{s}_{t+1})}{\partial \hat{s}_{t+1}}^{\top} \frac{\partial \hat{s}_{t+1}}{\partial a_t} \right\} \frac{\partial a_t}{\partial \eta} \quad (4)$$

Gradient ascent is also used to optimize the parameters of the critic ζ . The parameters are optimized to minimize the temporal difference error δ_t at time t , as shown in Equation 5. The equation of the temporal difference error and the update rule are shown in Equation 6.

$$\delta_t = \rho(s_t, a_t) + \gamma V_{\zeta}(s_{t+1}) - V_{\zeta}(s_t) \quad (5)$$

$$\zeta \leftarrow \zeta + \alpha_{cr} \delta_t \nabla_{\zeta} V_{\zeta}(s_t), \quad (6)$$

with α_{cr} the learning rate for the critic. When using eligibility traces with trace λ , the equation changes to

$$\zeta \leftarrow \zeta + \alpha_{cr} \delta_t z_t, \quad (7)$$

with $z_t = \lambda \gamma z_{t-1} + \nabla_{\zeta} V_{\zeta}(s_t)$ and $z_0 = 0$.⁷ This reduces to Equation 6 for $\lambda = 0$.

A similar method is used to improve the plant model. The parameters are optimized to minimize the error between the plant model prediction and the ground truth samples yielded by the plant.

$$\xi \leftarrow \xi + \alpha_{pm} (s_{t+1} - \hat{f}_{\xi}(s_t, a_t)) \nabla_{\xi} \hat{f}_{\xi}(s_t, a_t), \quad (8)$$

with α_{pm} the learning rate for the plant model.

The flow diagram of MLAC is shown in Figure 2. It shows the three components; the actor, the plant model and the critic.

Figure 2. Schematic flow diagram of the MLAC algorithm. Dashed lines are used to indicate training signals. Adapted from.²⁰

III.A. Local linear regression

MLAC represents the actor, critic and plant model by means of three separate FAs, which are all LLR FAs. Due to its local nature, LLR is able to learn rapidly from only few samples. Furthermore, it has been proven to work with MLAC for an inverted pendulum swing-up⁷ and a two-link acrobot swing-up system.²⁰

LLR keeps track of previously seen samples by storing them in a memory. The samples contain the function input x and output y and are represented as $p = [x|y]$. When querying the FA at the query point x_q , the FA selects k samples $[x_i|y_i]$ with $i \in \{1, 2, \dots, k\}$ nearest to the query point as measured in Euclidean distance. The samples will be arranged in the regression matrix X and will be appended with a bias of 1 to make the fit affine, as shown in Equation 9. The target values corresponding to the samples will be arranged in target matrix Y .

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_k \\ 1 & 1 & \cdots & 1 \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_k \end{bmatrix} \quad (9)$$

The linear regression is solved for the parameter vector $\hat{\beta}$ by the least squares method using the right inverse as shown in Equation 10.

$$\hat{\beta} = YX^\top(XX^\top)^{-1} \quad (10)$$

The prediction $\hat{y}_q$ for the query point x_q is then given by Equation 11. The process has been depicted in Figure 3.

Figure 3. Visualization of the local linear regression. Gray dots represent the k nearest neighbors.

In case of high simulation frequencies and/or low action range, the nearest neighbor samples may be close together, forcing the matrix $XX^\top$ to a singular one. Therefore the Moore-Penrose pseudo-inverse has been implemented instead of a regular inversion method. An alternative is to use ridge regression (also known as Tikhonov regularization) to regularize the parameters $\hat{\beta}$ fitted by the LLR by adding a quadratic penalty for the fitted parameters.²¹

III.B. Non-parametric MLAC

Originally MLAC has been designed to employ parametric FAs. FA updates are carried out by performing gradient ascent on the parameters describing the function. However, as LLR is a non-parametric FA, a few modifications need to be made.

In non-parametric MLAC, the LLR FA is updated in two ways; by adding new samples to the memory (1), and by adjusting the output values y of the samples (2). The plant model only employs the former method, as the samples come directly from the plant and are therefore considered ground truth. The actor and critic functions, however, use both methods, as no ground truth output y is known. These functions

bootstrap their values and therefore an adjustment is required. In case of the actor, the values of all nearest neighbors and the query point (if added) are updated by

$$a_i \leftarrow a_i + \alpha_{ac} \left\{ \frac{\partial \rho(s_t, a_t)}{\partial a_t}^\top + \gamma \frac{\partial V(\hat{s}_{t+1})}{\partial \hat{s}_{t+1}}^\top \frac{\partial \hat{s}_{t+1}}{\partial a_t} \right\} \quad \forall i \in \{1, 2, \dots, k\}, \quad (12)$$

where $\frac{\partial V(\hat{s}_{t+1})}{\partial \hat{s}_{t+1}}$ is derived from the estimated regression parameters $\hat{\beta}_{cr}$ in the critic and $\frac{\partial \hat{s}_{t+1}}{\partial a_t}$ from the estimated regression parameters $\hat{\beta}_{pm}$ in the plant model.

The critic uses an eligibility trace, so the update is done on all samples in the memory. At the start of each episode, all traces are set to zero. Whenever a sample p is visited (as state or neighbor), the trace $z_t(p)$ is set to 1. At each time step all traces z_t are updated with $z_t = \lambda \gamma z_{t-1}$. The values are incremented with $\alpha_{cr} \delta_t z_t(p_i)$, as shown in [Equation 13](#).

$$v_i \leftarrow v_i + \alpha_{cr} \delta_t z_t(p_i) \quad \forall i \in N_{cr}. \quad (13)$$

The derivations of the MLAC update procedures for LLR are described in [20](#).

III.C. Incremental plant model

In local FAs, a high input dimensionality can become an issue because of memory constraints. In the plant model, not only each state variable, but also each action variable increases the dimensionality, as opposed to the actor and critic. A control problem with 4 state variables and 2 action variables can already be difficult to approximate. Therefore a different implementation of the plant model has been developed, which is leaner in its variables. The proposed plant model is no longer a mapping from state and action to next state $\hat{s}' = \hat{f}(s, a)$, but rather a mapping from *rate* states and actions to state increments $\Delta \hat{s}'$ with respect to the current state. In this context the rate states are the first-order time derivatives of the position and attitude states.

Consider the following case. The state is described as $s_t = [x, \dot{x}, y, \dot{y}]^\top$ and the action as $a_t = [a_1, a_2]^\top$. The original plant model would approximate $\hat{x}', \hat{\dot{x}}', \hat{y}', \hat{\dot{y}}' = \hat{f}(x, \dot{x}, y, \dot{y}, a_1, a_2)$. The altered plant model approximation, however, approximates the state increments $\Delta \hat{x}', \Delta \hat{\dot{x}}', \Delta \hat{y}', \Delta \hat{\dot{y}}' = \hat{f}(\dot{x}, \dot{y}, a_1, a_2)$ with respect to the current state s_t . In total this allows the plant model to have two state variables less (x and y).

The logic behind this choice is that (1) only the difference between the next state and the current state needs to be computed, as all information of the current state is already known, and (2) the position or attitude is not required to predict the delta with respect to the next state $\Delta \hat{s}'$ as it is just the time-integral of the velocities and attitude rates. Therefore the plant model can be leaner.

The plant model serves two purposes. First, it is used to predict the next state. The plant model predicts the difference between the next state with respect to the current state. The current state can be added to form $\hat{s}_{t+1}$. The prediction of the next state is used by the critic to determine $\frac{\partial V}{\partial \hat{s}_{t+1}}$. The second purpose is to supply the policy gradient with the derivative of the next state with respect to the current action $\frac{\partial \hat{s}_{t+1}}{\partial a_t}$. The derivative can be split up into two parts as shown in [Equation 14](#). Because the derivative of the current state to the current action is zero, also this purpose of the plant model is still fulfilled with the proposed changes. Hence no other changes need to be made to MLAC to facilitate the incremental plant model.

$$\frac{\partial \hat{s}_{t+1}}{\partial a_t} = \frac{\partial (s_t + \Delta \hat{s})}{\partial a_t} = \frac{\partial s_t}{\partial a_t} + \frac{\partial \Delta \hat{s}}{\partial a_t} = \frac{\partial \Delta \hat{s}}{\partial a_t} \quad (14)$$

III.D. LLR Memory Management

During training the agent traverses many states and hence gathers many samples for storing in the memory of the LLR FA. If every sample would be added, the memory would grow indefinitely. Hence a form of memory management needs to be applied. Two forms of memory management are identified, one dealing with adding new samples to the memory and one with purging (removing) samples from the memory.

When encountering a new sample, the memory needs to decide whether to keep or discard the sample. For the actor and critic, a sample is stored when the average Euclidean *distance* to its nearest neighbors is more than a given tolerance $\epsilon_{ac}, \epsilon_{cr}$. For the plant model, a sample is added when the *prediction error* with respect to the ground truth exceeds a tolerance ϵ_{pm} . Alternative methods could be to keep it always (requires

a proper purging method), only when the distance to the closest nearest neighbor exceeds a tolerance ϵ , or when the new sample is expected to improve the local fit in future cases.

At the end of each episode, samples are purged from the memory if its maximum size has been violated. This is done by selecting samples randomly, whilst assigning a higher probability to older samples. The probability of being purged is equal to a samples age, divided by the total age of all samples in the memory, as shown in Equation 15. This method is used for all three memories. Alternative methods could discard samples fully randomly, or by age (deterministically).

$$\Pr_{\text{purge}}(p_i) = \frac{\text{Age}(p_i)}{\sum_{i=0}^N \text{Age}(p_i)} \quad (15)$$

III.E. Range searching for nearest neighbors

In finding the nearest neighbors for a query point, two challenges arise.

First, searching through the memory for the nearest neighbors is computationally expensive, especially for high-dimensional memories. For a few hundred samples, sorting all samples may still be feasible, but for an order of magnitude more, this becomes a very slow process. Hence a KD-Tree, a binomial search tree, is used to find the nearest neighbors in the memory.²² The tree is rebuilt each time a sample has been added or one or more have been removed.

Second, for samples with more than one dimension, the dimensions need to be scaled. If one variable x_1 has a smaller range of values than another x_2 , the FA is more likely to return samples that are close in x_1 than x_2 . Hence samples need to be scaled by a scale w in all dimensions before computing the Euclidean distance from a query point. Finding good values for the scale parameter of each state variable is a tedious process as the value has a strong effect on the performance of the agent.

IV. Flexible function approximators

An FFA is defined as a mathematical input/output function $y_1, y_2, \dots, y_m = f(x_1, x_2, \dots, x_n)$ that is able to

1. gain dependency on one or more new variables $x_{n+1}, x_{n+2}, \dots$,
2. gain one or more new outputs $y_{m+1}, y_{m+2}, \dots$,
3. lose dependency on one or more of the current variables $x_1, x_2, \dots, x_n$,
4. lose one or more of the current outputs $y_1, y_2, \dots, y_m$, and
5. change the FA order by adding or removing a higher-order term of the variables $x_1, x_2, \dots, x_n$ (this can be seen as a special case of points 1 and 3, where the variable is a non-linear combination of one or more variables in the FA).

Of paramount importance is that, when changing structure, prior knowledge needs to be exploited as a basis for further learning. Gaining dependency on new variables and adding outputs is referred to as state space or action space *expansion*. Losing dependency on variables and removing outputs is referred to as state space or action space *reduction*.

Some implementations of FFAs exist, such as Stepwise Regression (SR)^{23,24} and Adaptive Recursive Orthogonal Least-Squares (AROLS),²⁵ both structure selection and parameter estimation (SSPE) algorithms. The AROLS method has been used to identify the changing aerodynamic properties in passenger aircraft when experiencing structural failures. It selects new variables for its linear fit at each time step and removes redundant variables. They are however, unsuitable to RL applications. The episodic nature of RL control problems causes a discontinuity at the end of each episode. It may very well be that the selected structure at the end of an episode does not align with the desired structure at the start of the next episode. In other words, these SSPE algorithms are aimed at structure selection *during* episodes, whilst for flexible HDP structure selection needs to be done in between episodes.

Evolutionary Cascade Neural Networks are able to select inputs during training.²⁶ They start to learn with only one input node and then add new inputs as well as new hidden neurons to evolve. However, the number of output nodes is fixed from the start, making them unsuitable for application in flexible HDP.

As part of transfer learning, neural network architectures have been developed that support incremental learning of tasks, such as Progressive neural networks.²⁷ These architectures focus on forming internal, lateral connections in the hidden layers to support transfer of knowledge. None of them employs the addition (or removal) of input or output variables.

LLR FAs, however, support expansion and reduction, which comes down to mapping the samples in the memory to a higher and lower dimension respectively. All are based on mapping the samples in the LLR memory from $\mathbb{R}^n$ to $\mathbb{R}^{n+1}$, where n is the dimensionality of the LLR FA before expansion. Before expansion, the samples are described as $x = [x_1, x_2, \dots, x_n]^\top$. After expansion the samples are described as $x = [x_1, x_2, \dots, x_n, x_{n+1}]^\top$. Multiple ways of mapping samples in an LLR memory to a higher dimension can be used, such as assuming that all samples in the memory lie on the zero subspace of the new dimension x_{n+1} , hence for all samples in the memory $x_{n+1} = 0$. Another way would be to assign a random value to the new variable for each sample, e.g. using a Gaussian distribution. The way we employ here is to clone each sample several times and to draw the new value from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$. Distribution parameters μ and σ depend on the expected values of the new variable. The process is illustrated in Figure 4.

Figure 4. State space expansion by cloning each sample several times with values drawn from a Gaussian distribution.

IV.A. LLR FFA expansion in MLAC

Model-based actor-critics are made up of three components, the actor, the critic and the plant model. When expanding the FAs of MLAC, several changes need to be made at the same time to ensure that the policy gradient algorithm still functions properly. Consider a plant with n_s state variables $s = [s_1, s_2, \dots, s_{n_s}]^\top$ and n_a action variables $a = [a_1, a_2, \dots, a_{n_a}]^\top$. Hence $s \in \mathbb{R}^{n_s}$ and $a \in \mathbb{R}^{n_a}$. The critic will be a mapping from state to a scalar $\mathbb{R}^{n_s} \rightarrow \mathbb{R}$, as the rewards and the state values are always scalar. The actor will be a mapping from state to action $\mathbb{R}^{n_s} \rightarrow \mathbb{R}^{n_a}$. The plant model outputs a new state given a state and an action, so the mapping will be $\mathbb{R}^{n_s+n_a} \rightarrow \mathbb{R}^{n_s}$.

To ensure that the chain rule in the policy gradient (Equation 4) is mathematically sound, all three components need to work consistently with the same state and action variables.

When expanding the flexible MLAC with a state variable s_{n_s+1} , all three components require an update. Both the critic and actor gain a new input to their FA, whilst the plant model gets a new input as well as a new output. When adding a new action variable a_{n_a+1} , only the plant model gets a new input variable and only the actor expands with a new output variable.

IV.B. Limitations of flexible function approximators

A few limitations of FFA exist, imposed by the MLAC algorithm and by the LLR FA-type.

One of the main criteria of flexible MLAC is that all phases of the learning process must be an MDP or in case of a Partially-Observable MDPs (POMDPs) it should be well observable. Even converging to a near-optimal policy in POMDPs can be extremely challenging.²⁸ Concerning the POMDP, it proves difficult to define exactly how well observable it must be. The converse is that the less observable an MDP is, the poorer the experience is that the agent gains. Expanding that experience to a higher dimension can cause the agent to learn a poor, undesirable policy.

Furthermore, the policy gradient update of the MLAC algorithm also poses a restriction on the way that state-variables are added. One of the two purposes of the plant model is to supply the derivative

of the next state with respect to the current action. When taking a closer look at these variables, it is observed that the derivatives of the position and attitude state variables with respect to the current action are always (close to) zero. This makes sense as the values of these state variables are determined only by their time derivative, the integration method and the time step, so the action is not directly involved here. Consequently, no information to update the policy, can be exchanged from the critic to the actor through position and attitude state variables. This entails that for a position or attitude state variable to be added, its time derivative (rate) needs to be added as well.

Finally, expanding the plant model requires special attention. As opposed to the actor and the critic, its values are not updated by adjusting the output value as all samples come directly from the plant and are therefore considered ground truth. Mapping these samples to a FA with a higher dimensionality, is therefore invalid, as we may very well create a false ground truth. Therefore, when adding a new state, the plant model is emptied completely. After expanding the FFA with a state variable, a few episodes need to be run without updating the actor (thus using the old policy), such that the plant model has opportunity to recuperate its experience.

V. Experiment set-up

The flexible MLAC algorithm is tested on a 2D hovering problem for a quad-rotor. This section describes the experiment set-up.

V.A. Quad-rotor system

The quad-rotor system is a state space model that yields the next state based on a state and action. It is modeled after the Parrot AR 2.0 drone and has been implemented for RL control previously.²⁹ The model is considered perfect and noise-less and will henceforth be referred to as the *plant*.

The quad-rotor has four engines placed symmetrically around its center of gravity. It is able to rotate around its main axes by increasing the thrust of two of the four rotors. Translations are performed by rotating the quad-rotor to the desired attitude and increasing the collective power of all rotors. The quad-rotor has a plus-configuration meaning that the positive x_b -axis coincides with the arm from which engine 1 is suspended. The system and its body reference system are shown in Figure 5.

Figure 5. Quad-rotor configuration and body axis reference system.

The force vector takes into account gravity, body drag, body lift, and of course the rotor forces, for which a static relation between the thrust, torque and rotor rotational velocity is used. In terms of moments the plant takes into account the fuselage moment due to frame geometry, the moment due to rotor torques, blade flapping effects and the gyroscopic moment of each rotor. For a more in-depth explanation of the dynamics, the reader is referred to.²⁹

To create a 2-dimensional version of the plant, the yaw angle ψ and moment M_ψ , roll angle ϕ and moment M_ϕ as well as the sway force F_y and position y are constrained to zero. The system state at time t is therefore represented by the state vector $s_t = [x, z, \dot{x}, \dot{z}, \theta, \dot{\theta}]^\top$. Note that the positions x, z in the state vector are in the inertial reference system (see Appendix VIII).

Some adjustments have been made to make the control easier and to reduce the dimensionality of the state space. For one the assumption is made that the rotor speed of each rotor can be controlled directly

and instantaneously. Secondly, the actions available to the agent are mapped rotor speeds. Action a_1 is the collective rotor speed of all rotors, and action a_2 represents the rotor speed which is added to the front rotor and subtracted from the rear one. Hence the action a_1 effectively controls translation along the body z_b -axis, and action a_2 controls the pitch angle around the body y_b -axis. The action at time t is represented as $a_t = [a_1, a_2]^\top$.

The rotor speed range for the Parrot AR 2.0 drone is $[0, 3400]$ revolutions per minute (RPM). The actions are mapped such that an action of 0 corresponds to the lift-off rotor speed $\omega_{\text{lift-off}} = 1920$ RPM (given the vehicle configuration), +1 corresponds to the maximum of 3400 RPM and the minimum -1 hence corresponds to 440 RPM. The mapping from action to rotor speed can be summarized by

$$\omega_1 = \text{sat}(\omega_{\text{lift-off}} + \kappa_a(a_1 + a_2)), \text{ and} \quad (16)$$

$$\omega_2 = \text{sat}(\omega_{\text{lift-off}} + \kappa_a(a_1 - a_2)), \quad (17)$$

with action scale parameter $\kappa_a = 1480$ RPM, and *sat* a function that saturates the rotor speeds between $[0, 3400]$ RPM.

V.B. Goal and reward function

The goal for the agent is to hover at the desired altitude $z_i = -10\text{m}$ with all velocities equal to zero, when starting each episode at $[x_i, z_i] = [0, -9]\text{m}$, where i denotes the inertial reference frame. The altitude have been chosen to minimize interference with the ground. Hence for each episode $s_0 = [0, -9, 0, 0, 0, 0]^\top$ and $s_{\text{desired}} = [0, -10, 0, 0, 0, 0]^\top$. Note that a negative value for z_i corresponds to an altitude above ground, as the z_i -axis points downward. The lateral position is not observed, and hence the agent is not penalized for any non-zero x or $\dot{x}$.

A continuous quadratic error reward function has been chosen, which yields a reward based on the quadratic error of the state with respect to the desired state, and penalizes any non-zero action. Actions are penalized to drive the agent to only use them when necessary, as in UAVs battery power is limited. It also helps to regularize and smoothen the action signal. The instantaneous reward function is defined as

$$\rho(s_t, a_t) = -\frac{1}{2} (e_t^\top W_s e_t + a_t^\top W_a a_t), \quad (18)$$

with $e_t = s_{\text{desired}} - s_t$.

In Equation 18, W_s and W_a are diagonal matrices with the weights for the state errors and the action on the diagonal. Their values can be found in section VIII.

V.C. Training procedure

The agent will be trained by having it interact with the plant for 150 episodes, each one lasting $T = 3$ seconds. All episodes will be initialized in the point s_0 . The agent will start the training procedure without any experience, e.g. no samples in the memories.

Two agents are trained for the same purpose, to hover at the desired state s_{desired} :

1. An **SFA agent** that contains the state variables $s = [z, \dot{z}, \theta, \dot{\theta}]^\top$ and action variables $a = [a_1, a_2]^\top$ from the start in its FAs and reward function.
2. An **FFA agent** that starts with a limited state and action space and gradually gains access to more variables. The first 75 episodes (phase one) its state space consists of $s_{\text{phase1}} = [z, \dot{z}]^\top$ and its action space is $a_{\text{phase1}} = [a_1]$. After 75 episodes phase two starts, and the second action a_2 , pitch angle state θ and pitch rate $\dot{\theta}$ are added manually to form the new state space $s_{\text{phase2}} = [z, \dot{z}, \theta, \dot{\theta}]^\top$ and action space $a_{\text{phase2}} = [a_1, a_2]^\top$. The method used to expand the state space is in both cases the clone method with a Gaussian distribution. When expanding the state space with θ and $\dot{\theta}$ all samples are cloned 3 times. The parameters can be found in section VIII. Rewards are only given for the state and action variables which are observable by the agent.

As the algorithm uses stochastic behavior to drive exploration, the end result will also be stochastic. Therefore both agents are trained 64 times, and the result will be evaluated by the distribution over these 64 trials.

At the very start, when less than k samples are present in the memories, the LLR FAs behave slightly differently. The critic will predict $\hat{y}_q = 0$ and yield $\hat{\beta} = 0$. The actor does not yield a zero action, but rather a random action according to $\mathcal{N}(0, \frac{1}{3}a_{\text{limit}})$, which is required as the update of the actor depends on the derivative $\frac{\partial \hat{s}_{t+1}}{\partial a_t}$. The derivative will always be zero if all the samples in the plant model have zero as their action value. Hence, the symmetry needs to be broken to get the system to start learning. The plant model predicts the input state when its memory contains less than k samples.

To make a fair comparison between the SFA and FFA agent, a closer look needs to be taken at the learning parameters. As these strongly drive the final performance of the agent, they need to be chosen carefully. The fairest way to go about this, is to optimize the learning parameters of both agents separately using the same goal and the same set of candidate parameters. Both learning parameter sets have been optimized using Bayesian optimization with a Gaussian process, so the function value is assumed to follow a multivariate Gaussian distribution over the parameter values. The covariance of the values is given by the Gaussian process kernel between the parameters. The next combination of parameters is chosen by evaluating the negative expected improvement acquisition function over the Gaussian prior.

The pseudo code for the training is shown in [section VIII](#).

V.D. Exploration

For policy gradient methods, no exploration is required to tune the actor, as this is optimized by gradient ascent. However, exploration is required for exploring the value function. The agent needs to end up in different states to evaluate the value function there. Therefore, exploration has been such that transitioning from one state to another the action is occasionally perturbed, causing the agent to end up in another state than it intended.

The perturbation is applied at a particular interval j , meaning that exploration is performed every j -th time step. For the SFA agent $j = 2$ for all episodes. For the FFA agent $j_1 = 2$ for phase one and $j_2 = 3$ for the second phase.

Perturbations are distributed as a Gaussian process $\mathcal{N}(\mu, \sigma)$, where $\mu = 0$ due to the symmetry in the action space, and $\sigma = \frac{1}{3}a_{\text{limit}}$, such that 99% (3σ) of the perturbations fall within the actuator limits. Evidently the actions are saturated to the controller limits after perturbation.

VI. Results

This section discusses the results of the training process of the SFA and FFA agent, and gives a comparison of the two. The agents are only evaluated by means of their policy and the reward gathered during the episodes. Accuracy of the plant model is not regarded, as having a very accurate plant model does not necessarily mean that the system will train fast or that the final policy will be optimal.

The SFA agents learning process is shown in [Figure 6](#). From the increasing median reward, it can be seen the agents learn something and improve their policy. However, the reward signal is very noisy, and is far from robust. Furthermore, by observing the jumps in the median in the last 20 episodes, it has not converged properly. It appears it is difficult to train a SFA agent to achieve a consistent, good policy for the the 4-dimensional quad-rotor control problem. It is plausible that further tuning of the learning parameters could still improve this result.

In [Figure 7](#) the learning process of the FFA agent is shown. The learning starts out quickly reaching a decent policy in about 20 episodes. The agent continues to improve, mainly by learning the correct moment to reverse thrust when approaching the desired altitude. In episode 75 the action a_2 , and state variables θ and $\dot{\theta}$ are added. The next 3 episodes, no updates are done on the policy, but as the actions get perturbed for exploration, the reward decreases drastically. However, within another 30 episodes, the agent is able to recover and carry out the task within a larger state and action space. The cumulative reward does not reach the previous optimum, which can be attributed to a few reasons. First, two extra state variables and one additional action variable is added, for which (negative) rewards are given during the episode. Second, the actions are now perturbed in a second dimension as well, causing additional reward. Lastly, the higher-dimensional action and state spaces cause a larger complexity to make proper linear fits, hence the policy quality deteriorates.

The FFA agent learns a better policy than the SFA agent, as can be seen by the rewards in the last episodes. Where the SFA agent reaches a policy yielding a median reward of about -800 during an episode,

Figure 6. Cumulative rewards gathered during the training of 64 SFA MLAC agents.

Figure 7. Cumulative rewards gathered during the training of 64 FFA MLAC agents. At the 75th episode, the additional state variables and action are added.

the FFA agent has found a policy yielding a median reward per episode of around -240. It seems that the agent benefits from learning initially in a simpler environment and porting that knowledge to the actual control problem. However, it is not said that the FFA method is better than the SFA method. Even though much time has been spent on finding good learning parameters for both agents, no guarantee can be given that no better parameter set is possible.

To give an idea of the policy that is achieved by the FFA agent, a simulation has been run of which the states and actions are shown in [Figure 8](#) and [Figure 9](#). The former shows by a visualization of the motion, where each third time step is frozen. Less opaque lines are further in the past. The latter figure shows the states and actions over time. The agent is able to keep the quad-rotor hovering at the desired altitude with minor pitch angles. It drifts off in the lateral (x) dimension, which happens as that state variable is not observable by the agent.

Figure 8. Motion visualization of a single simulation, where each third time step is frozen. More opaque lines are further in the past.

Figure 9. Quad-rotor states and the actions chosen by the learned policy.

VII. Conclusion

In this paper, a flexible version of the MLAC algorithm is proposed. Flexible MLAC shows that it is possible to train an RL agent on part of the action and state space and then extend that knowledge to higher action and state spaces. The actor, critic and plant model in MLAC have been represented by LLR FAs. State space and action space expansion is achieved by mapping the samples in the LLR memories to a higher dimension, where the value of the new variable is drawn from a Gaussian distribution. Each sample is cloned a few times, with different values for the new variable to ensure higher coverage in the new dimension. By adjusting the output values of the samples in the memory, the agent is able to improve the knowledge that was mapped to the higher dimension. The FFA agent outperforms the SFA agent in a task of hover control for a 2D quad-rotor. Not only does it learn faster, it also achieves a more accurate policy. However, as the process of finding good learning parameters is difficult, no definitive conclusion is given on which method performs better. Nonetheless, it can be concluded that it is possible to train flexible model-based RL agents in a gradual manner. This is more similar to how humans learn when initially constraining the state and action space of a control problem.

A few limitations of flexible MLAC have been observed. For one, due to the nature of the LLR, when adding state variables the plant model needs to be emptied, discarding valuable experience. Furthermore, position and attitude variables can only be added together with their time derivative. This is required as the derivative of a position or attitude with respect to an action is always zero and hence no policy gradient can be determined. Finally, all phases in which the FFA agent trains, need to be MDPs or POMDPs that are well observable.

VIII. Recommendations for future research

The method behind FFAs has been shown to work and gain initial success. There are, however, many points on which the method can be improved. First, the method can be expanded to global FAs, to see whether better results can be achieved there. A parametric FA has the advantage that it does not need to lose the prior knowledge in the plant model when changing structure, as is the case with the LLR FAs. This likely speeds up learning as no in-between episodes without actor updates are required to get the plant model up to speed again. Furthermore, the added state and action variables are currently handpicked and added manually. Ideally the algorithm is able to know when it is not improving anymore and try out candidate state or action variables from a predefined pool. This requires a heuristic that quickly assesses the effect of a new variable. To enable this, a method for forgetting irrelevant state or action variables therefore needs to be developed as well. As a last recommendation, the way that MLAC uses the policy gradient requires the state space to contain the time derivative (rate) of each position and attitude variable. This means that new state variables need to be added in pairs. The performance of an FFA is likely to improve when state variables can be added one-by-one as the cloning of samples is then only done in one dimension instead of two.

References

- ¹Sutton, R. S. and Barto, A. G., *Reinforcement Learning: An Introduction*, MIT Press, 1998.
- ²Singh, S. P., “Transfer of Learning by Composing Solutions of Elemental Sequential Tasks,” *Machine Learning*, Vol. 8, No. 3, 1992, pp. 323–339.
- ³Thrun, S., “Is Learning The n-th Thing Any Easier Than Learning The First?” *Advances in Neural Information Processing Systems*, 1996, pp. 7.
- ⁴Taylor, M. E. and Stone, P., “Transfer Learning for Reinforcement Learning Domains : A Survey,” *Journal of Machine Learning Research*, Vol. 10, 2009, pp. 1633–1685.
- ⁵Lu, J., Behbood, V., Hao, P., Zuo, H., Xue, S., and Zhang, G., “Transfer learning using computational intelligence: A survey,” *Knowledge-Based Systems*, Vol. 80, 2015, pp. 14–23.
- ⁶Ring, M. B., “CHILD: A First Step Towards Continual Learning,” *Machine Learning*, Vol. 28, 1997, pp. 77–104.
- ⁷Grondman, I., Vaandrager, M., Busoniu, L., Babuška, R., and Schuitema, E., “Efficient Model Learning Methods for ActorCritic Control,” *IEEE Transactions on Systems, Man, and CyberneticsPart B: Cybernetics*, Vol. 42, No. 3, 2012, pp. 591–602.
- ⁸Atkeson, C. G. and Santamaria, J., “A comparison of direct and model-based reinforcement learning,” *Proceedings of International Conference on Robotics and Automation*, Vol. 4, No. 4, 1997, pp. 3557–3564.
- ⁹Lin, L.-j., “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching,” *Machine Learning*, Vol. 8, No. 3-4, 1992, pp. 293–321.

- ¹⁰Moore, A. W. and Atkeson, C. G., "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time," *Machine Learning*, 1993, pp. 103–130.
- ¹¹Sutton, R. S., "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, Vol. 2, No. 4, 1991, pp. 160–163.
- ¹²Powell, W. B., "What You Should Know About Approximate Dynamic Programming," *Naval Research Logistics*, Vol. 55, No. April 2007, 2006, pp. 541–550.
- ¹³Watkins, C. J. C. H., *Learning From Delayed Rewards*, Ph.D. thesis, University of Cambridge, England, 1989.
- ¹⁴Satinder, S. P. and Sutton, R. S., "Reinforcement Learning with Replacing Eligibility Traces," *Machine Learning*, Vol. 22, 1996, pp. 123–158.
- ¹⁵Rummery, G. A. and Niranjan, M., "On-line Q-learning using connectionist systems," Tech. Rep. September, Cambridge University Engineering Department, Cambridge, England, 1994.
- ¹⁶Gullapalli, V., "A stochastic reinforcement learning algorithm for learning real-valued functions," *Neural Networks*, 1990.
- ¹⁷Williams, R. J., "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, Vol. 8, No. 3, 1992, pp. 229–256.
- ¹⁸Lendaris, G. G., "A retrospective on Adaptive Dynamic Programming for control," *Proceedings of the International Joint Conference on Neural Networks*, 2009, pp. 1750–1757.
- ¹⁹Buşoniu, L., Schutter, B. D., and Babuška, R., "Approximate dynamic programming and reinforcement learning," *Automation and Control Engineering Series*, CRC Press, 2010.
- ²⁰Grondman, I., *Online Model Learning Algorithms for Actor-Critic Control*, Ph.D. thesis, Delft University of Technology, 2015.
- ²¹Tikhonov, A. N., "Solution of Incorrectly Formulated Problems And Regularization Method," *Doklady Akademii Nauk SSSR*, 151, Vol. 151, No. 3, 1963, pp. 501–504.
- ²²Bentley, J. L. and Friedman, J. H., "Data Structures for Range Searching," *ACM Computing Surveys*, Vol. 11, No. 4, 1979, pp. 397–409.
- ²³Draper, N. and Smith, H., *Applied Regression Analysis*, Wiley, New York, 1981.
- ²⁴Klein, V., Batterson, J. G., and Murphy, P. C., "Determination of airplane model structure from flight data by using modified stepwise regression," Tech. rep., 1981.
- ²⁵Lombaerts, T., van Oort, E., Chu, Q. P., Mulder, J. A., and Joosten, D., "Online Aerodynamic Model Structure Selection and Parameter Estimation for Fault Tolerant Control," *Journal of Guidance, Control, and Dynamics*, Vol. 33, No. 3, 2010, pp. 707–723.
- ²⁶Schetinin, V., "A learning algorithm for evolving cascade neural networks," *Neural Processing Letters*, 2003.
- ²⁷Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R., "Progressive Neural Networks," *arXiv*, 2016.
- ²⁸Ng, A. Y., *Shaping and policy search in reinforcement learning*, Ph.D. thesis, University of California, Berkeley, 2003.
- ²⁹Molenkamp, D., van Kampen, E.-J., de Visser, C. C., and Chu, Q. P., "Intelligent Controller Selection for Aggressive Quadrotor Manoeuvring," *AIAA SciTech Forum*, 2017, pp. 1–15.
- ³⁰Raza, S. A. and Gueiaeb, W., "Intelligent Flight Control of an Autonomous Quadrotor," *Motion Control*, , No. 1, 2010, pp. 245–264.

Appendix: Learning parameters

Table 1. Generic system and learning parameters.

Parameter	Symbol	Value
Number of trials for each agent	n_{trials}	64
Number of training episodes per trial	n_{episodes}	150
Episode length	T	3 [s]
Time step	Δt	1/50 [s]
Eligibility trace of critic	λ_{cr}	0.75
Discount rate	γ	0.948
Reward weights for action	W_a	[3.55, 4.98]
Reward weights for state	W_s	[0, 10, 0, 0.45, 7, 0.3]
Distribution for action perturbation	$\mathcal{N}(\mu, \sigma)$	$\mathcal{N}(0, \frac{1}{3}a_{\text{limit}})$
Maximum age of samples in memory	-	45 [s]
Scales for state variables in KD-Tree	w_s	[2.72, 5, 2.30, 6.35]
Scales for action variables in KD-Tree	w_a	[1, 0.76]

Table 2. System and learning parameters for SFA and FFA agents.

Parameter	Symbol	SFA agent	FFA agent
Actor/critic memory size (phase 1)	$N_{\text{ac/cr}}$	6600	1000
Actor/critic memory size (phase 2)	$N_{\text{ac/cr}}$	-	9500
Plant model memory size (phase 1)	N_{pm}	6400	350
Plant model memory size (phase 2)	N_{pm}	-	6500
Number of nearest neighbors for actor (phase 1)	$k_{\text{ac},1}$	21	16
Number of nearest neighbors for actor (phase 2)	$k_{\text{ac},2}$	-	25
Number of nearest neighbors for critic (phase 1)	$k_{\text{cr},1}$	21	20
Number of nearest neighbors for critic (phase 2)	$k_{\text{cr},2}$	-	30
Number of nearest neighbors for plant model (phase 1)	$k_{\text{pm},1}$	17	9
Number of nearest neighbors for plant model (phase 2)	$k_{\text{pm},2}$	-	27
Exploration interval (phase 1)	j_1	2	2
Exploration interval (phase 2)	j_2	2	3
Tolerance for adding samples (per dimension)	ϵ_{ac}	7.57×10^{-4}	2.08×10^{-3}
Tolerance for prediction error	ϵ_{pm}	1.21×10^{-4}	4.98×10^{-7}
Learning rate of actor	α_{ac}	0.189	0.15
Learning rate of critic	α_{cr}	0.0307	0.27
Standard deviation for expansion of θ	σ_{θ}	-	0.11
Standard deviation for expansion of $\dot{\theta}$	$\sigma_{\dot{\theta}}$	-	0.76
Number of clones upon adding θ	-	-	3
Number of clones upon adding $\dot{\theta}$	-	-	3

Appendix: Pseudo code

Algorithm 1 Algorithm for flexible LLR MLAC.

Require: γ, λ and α for actor, critic and plant model

```

1: Initialize function approximators
2: for episode in  $n_{\text{episodes}}$  do
3:   if expand then ▷ expand function approximator
4:     for  $p$  in  $\{M_{\text{ac}}, M_{\text{cr}}, M_{\text{pm}}\}$  do
5:        $p \leftarrow \text{expand}(p, s_{n+1})$  ▷ using chosen expansion strategy
6:   for  $M$  in  $\{M_{\text{ac}}, M_{\text{cr}}, M_{\text{pm}}\}$  do ▷ purge samples from memories
7:     while  $N_M > N_{M,\text{max}}$  do purge sample  $p$  ▷ according to purge strategy
8:    $t \leftarrow 0$ 
9:    $z_t = 0$ 
10:  choose  $s_0$ 
11:  loop ▷ run episode
12:     $a_t \leftarrow \pi(s_t)$  ▷ compute action
13:    Measure  $s_{t+1}, r_{t+1}$ 
14:     $\delta_t \leftarrow r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$  ▷ compute TD error
15:    add  $[s_t | V(s_t)]$  to  $M_{\text{cr}}$  ▷ add sample to critic memory
16:    for  $p$  in  $K_{\text{cr}}$  do ▷ set trace of nearest neighbors to 1
17:       $z_t(p) \leftarrow 1$ 
18:    for  $p$  in  $M_{\text{cr}}$  do ▷ update values of critic nearest neighbors
19:       $V(s_p) \leftarrow V(s_p) + \alpha_{\text{cr}} z_t(p) \delta_t$ 
20:    add  $[s_t, a_t | s_{t+1}]$  to  $M_{\text{pm}}$  ▷ add sample to plant model memory
21:    add  $[s_t | a_t]$  to  $M_{\text{ac}}$  ▷ add sample to actor memory
22:    for  $p$  in  $K_{\text{ac}}$  do ▷ update actions of actor nearest neighbors
23:       $a_p \leftarrow a_p + \alpha_{\text{ac}} \left\{ \nabla_a \rho(s_t, a_t)^\top + \gamma \nabla_s V(\hat{s}_{t+1})^\top \nabla_a \hat{f}(s_t, a_t) \right\}$ 
24:       $z_t(p) \leftarrow \lambda_{\text{cr}} \gamma z_{t-1}(p)$  ▷ update eligibility traces
25:      for  $p$  in  $\{M_{\text{pm}}, M_{\text{cr}}, M_{\text{pm}}\}$  do ▷ age samples in memories
26:        Age( $p$ )  $\leftarrow$  Age( $p$ ) + 1
27:       $\Delta a_t \sim \mathcal{N}(0, \sigma)$  ▷ explore
28:       $s_{t+1} \leftarrow \text{Plant}(s_t, a_t + \Delta a_t)$  ▷ transition to next state
29:       $t \leftarrow t + 1$ 

```

Appendix: Reference frames

Three main frames of references $\mathcal{F}$ are defined for quad-rotors.³⁰ They have been depicted in **Figure 10**.

- The inertial frame $\mathcal{F}_i = (x_i, y_i, z_i)$ is an earth-fixed coordinate system with the origin at the base station. The x -axis points North, the y -axis East and the z -axis towards the center of the earth.
- The body frame of reference $\mathcal{F}_b = (x_b, y_b, z_b)$ has its origin at the center of gravity of the quad-rotor. The x -axis coincides with the arm of the quad-rotor structure with the front motor. The y -axis is along the arm of the right motor. The z -axis is determined using the right-hand rule from $z_b = x_b \times y_b$ where $\times$ denotes the cross-product.
- The vehicle frame $\mathcal{F}_v = (x_v, y_v, z_v)$ is an inertial frame of which the origin is located at the center of gravity of the quad-rotor. All axes are parallel to their counterparts in the inertial frame $\mathcal{F}_i$.

Figure 10. Three reference frames for quad-rotors. Taken from³⁰