CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	On-line reinforcement learning for trajectory following with unknown faults
Authors	Sohége, Yves;Provan, Gregory
Publication date	2018-12
Original Citation	Sohége, Y. and Provan, G. (2018) 'On-line reinforcement learning for trajectory following with unknown faults', Proceedings of the 26th AIAI Irish Conference on Artificial Intelligence and Cognitive Science (AICS 2018), Dublin, Ireland, 6-7 December, pp. 1-12. Available at: http://ceur-ws.org/Vol-2259/aics_27.pdf (Accessed: 28 January 2019)
Type of publication	Conference item
Link to publisher's version	http://ceur-ws.org/Vol-2259/aics_27.pdf, http://ceur-ws.org/Vol-2259/
Rights	© 2018, the Authors. Copying permitted for private and academic purposes.
Download date	2026-11-15 17:43:00
Item downloaded from	https://hdl.handle.net/10468/7366

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

On-line Reinforcement Learning for Trajectory Following with Unknown Faults

Yves Sohége[1 1][0000−0002−3942−0454] and Gregory Provan[2 1][0000−0003−3678−046X]

Insight-Centre for Data Analytics, University College Cork, Cork, Ireland
yves-sohege@insight-centre.ie, g.provan@cs.ucc.ie

Abstract. Reinforcement learning (RL) is a key method for providing robots with appropriate control algorithms. Controller blending is a technique for combining the control output of several controllers. In this article we use on-line RL to learn an optimal blending of controllers for novel faults. Since one cannot anticipate all possible fault states, which are exponential in the number of possible faults, we instead apply learning on the *effects* the faults have on the system. We use a quadcopter path-following simulation in the presence of unknown rotor actuator faults for which the system has not been tuned. We empirically demonstrate the effectiveness of our novel on-line learning framework on a quadcopter trajectory following task with unknown faults, even after a small number of learning cycles. The authors are not aware of any other use of on-line RL for fault tolerant control under unknown faults.

Keywords: Reinforcement Learning · Fault-tolerant Control · Quadcopter control

1 Introduction

One of the most important uses of reinforcement learning (RL) is for controlling robots, using reinforcement learning control (RLC). It has been shown, e.g., [1] that RLC can provide a model-free method for learning control of a robot, e.g., a quadrotor.

Model-free RLC assumes that the control system starts with no model and solves the Bellman equation based on running experiments with appropriate rewards, to create a matrix of values that serves as the model. Although model-free RLC can prove accurate, its main disadvantage is a long convergence time. Since the policy space for robotic interactions can be extremely large, RLC requires a large number of iterations to achieve convergence. In addition, if the plant is unstable (as is that of a quadcopter), or safety is an issue, using RLC can prove difficult in practice.

The alternative approach is to use model-based RLC, which is also known as iterative learning control (ILC). ILC refines the reference or input signals of a

CEUR-WS.org/Vol-2259/aics_27.pdf

desired maneuver based on data from previous executions. This can be used to update model parameters or extend an existing model.

We assume that we have modelled the quadcopter with a linear approximation of the underlying non-linear flight dynamics. Further, we assume that unmodeled dynamics (in our case, faults) can be represented as linear multiplicative term to the actuation dynamics. Given planned trajectory inputs $\boldsymbol{u}_k$, each ILC iteration can be decomposed into two steps: (1) disturbance estimation, where a Kalman filter computes the current estimate of the disturbance ; and (2) input update,where we compute an improved quadrotor input, $\boldsymbol{u}_{k+1}$. Using this framework, the input can be abstracted at any level; e.g., from robot thrust and angular velocities [2] to the position commands for trajectory following [3–5]. Experiments show that few iterations (on the order of 10-20) are needed to characterize repeatable disturbances and improve tracking performance.

The majority of applications of RL assume that learning is done off-line. However, there are many situations in which a robot will encounter novel situations and needs to adapt to those situations. We address that scenario in this article. In particular, we examine a quadcopter that has been pre-programmed with a set of controls and uses control blending to operate within a known control environment. However, for novel scenarios the robot must adapt. We introduce novel actuator faults into a quadcopter, and use ILC to learn new control laws.

We have defined the quadcopter to have a hierarchical control architecture. The lower-level controllers use PID methods to control each of the quadcopters three axis of movement. The higher-level controller uses blends of the lower-level controllers to control flight trajectories. We have pre-defined a set of control laws for nominal and fault modes. We then subject the quadcopter to unseen fault and then allow the quadcopter to repeat the novel conditions for the unseen faults to learn new high-level control laws.

This article proposes the first use of ILC for learning novel fault-tolerant control laws. We empirically demonstrate that we can learn new controls from a small number of learning trials.

2 Related Work

This work builds on extensive prior work in RL, fault-tolerant control (FTC) and Fault Detection and Isolation (FDI).

A significant body of work exists for RL for robotics applications, dating from [1]. This work is related to work on trajectory following, e.g., [3–5]. For this class of application, several instances of learning quadcopter control have been achieved [6]; however we are not aware of prior work that uses Reinforcement Learning to learn the optimal blending of controllers and achieve fault tolerant control.

In the area of FTC [7], a significant body of work has been developed and applied to real-world systems. [8] presents a recent overview of FTC, and [9] presents FTC with relation of system safety. Traditional methods for FTC employ a bank of observers coupled with dedicated controllers, and perform dis-

crete switching. This approach enables designers to tune the system to dedicated faults, but the speed of the system hinges on the speed of FDI. More recent approaches use mixing controllers, e.g., [10, 11], which blend the outputs of multiple controllers and are less reliant on FDI.

Fault-Tolerant Control (FTC) can be divided into two types, passive and active. Passive fault tolerant controllers are designed off-line against predefined models for certain operating conditions and have no ability to react to unanticipated faults. Passive FTC enables fast adaptation to faults, within the predefined operating conditions. Active FTC uses on-line data to reconfigure the controller to stabilise the plant. For a comprehensive study between the two approaches see [12]. Both active and passive FTC rely on specifying the space of faults that the system will encounter. For passive FTC, approaches such as blending of controllers tuned to nominal and failure modes are used to maintain system stability, e.g., [13]. Analogously, active FTC can rely on being able to detect pre-specified faults, such as using a bank of observers, with each observer tuned to a particular fault, e.g. [14]. For complex systems, it is impossible to pre-specify all faults, since there are too many fault combinations to consider, and it may be impossible to know all possible faults a priori. As a consequence, it is imperative that a system designer understand the space of possible faults and their impact on a system. Very little work has been conducted on exploring the space of faults and their impact on active vs passive FTC.

3 Reinforcement Learning

Reinforcement learning (RL) [15] is a technique for learning control actions that are optimal for particular states, using interactions of an agent with the environment in which the agent obtains rewards for actions. Reinforcement learning is typcially formalized as a Markov decision process (MDP), which is a tuple $M = \langle S, U, T, R, \gamma \rangle$, where

- S is the set of possible world states,
- U is the set of possible control actions,
- T is a transition function $T : S \times U \to P(S)$,
- R is the reward function $R : S \times U \to R$,
- and γ is a discount factor such that $0 \leq \gamma \leq 1$.

Reinforcement learning learns a policy $\mathit{\Pi} : S \to U$, which defines which actions should be taken in each state. Q-learning [16] is a model-free reinforcement learning technique that uses a Q-value $Q(s, u)$ to estimate the expected future discounted rewards for taking action u in state s. At each step, Q-learning applies an update equation for $Q(s, u)$ given by

$$Q(s_t, u_t) \leftarrow Q(s_t, u_t) + \alpha \left(r_{t+1} + \gamma \max_u Q[s_{t+1}, u_t] - Q[s_t, u_t] \right)$$

where r_{t+1} is the reward observed after performing action u_t in state s_t, α is the learning rate ($0 \leq \alpha \leq 1$), and s_{t+1} is the state that the agent transitions to

after performing action u_t. After $Q(s_t, u_t)$ converges, the optimal action for the agent in state s_t is $\arg\max_u Q(s_t, u)$.

In Q-learning and related algorithms, an agent maintains a table of $Q[S, U]$ based on its history of interaction with the environment. An experience $\langle s, u, r, s' \rangle$ provides one data point for the value of $Q(s, u)$. The data point is that the agent received the future value of $r + \gamma V(s')$, where $V(s') = \max_{u'} Q(s', u')$; this is the actual current reward plus the discounted estimated future value. This new data point is called a return. The agent can use the temporal difference equation to update its estimate for $Q(s, u)$:

$$Q[s, u]Q[s, u] + \alpha(r + \gamma \max_{u'} Q[s', u'] - Q[s, u])$$

or, equivalently,

$$Q[s, u](1 - \alpha)Q[s, u] + \alpha(r + \gamma \max_{u'} Q[s', u']).$$

4 RL Results

This section presents a summary of our results. We have shown that RL can be used to dynamically learn how to stabilise a robot's trajectory given previously unseen faults. We used a quadcopter to demonstrate how path-following deviations can be reduced following learning epochs.

We will firstly present the final Q-Matrix (Figure 1) that was learned. Positive and negative Q-values represent good and bad performance respectively. There is a large amount of negative values which can be attributed to our reward function, see Section 5.1. We say the matrix has converged perfectly if only a single positive Q-Value exists per row. This is not evident in this matrix but with adjustments to the reward function and enough learning epochs this matrix would converge. Another interesting point to note is the magnitude of the Q-Values. The first row has significantly higher values than the rest. This is because the first row represents the lowest deviation rate and this row will get trained for every fault since for the deviation rate to increase to the higher partitions it must first go through first partition. To see the improvements in trajectory following performance for a quadcopter, we compared the RLC with the original quadcopter configuration. As far as the authors are aware there is currently no baseline controller for unknown faults to compare against. In future we intend to run comparisons against other FTC architectures. We use total trajectory deviation in centimetres to gauge how much the tracking accuracy improved. Table 1 shows this comparison for several magnitudes of unknown rotor faults. We denote the number of learning cycles of RLC using μ. We noticed a 63-75% decrease in the trajectory deviation after a small number of learning cycles.

Figure 2 shows the time it took to re-stabilize the quadcopter along its trajectory with acceptable deviation after each completed learning cycle. It is clearly visible that after a small number of learning iterations the time to re-stabilize drastically reduces but never converges to 0. The spikes in the graph show an increase in the magnitude of a benchmark fault induced.

	1	2	3	4	5	6	7	8	9	10
1	5.5512	-14.4008	-7.4802	-20.3058	-21.5061	-11.5752	-19.8285	-18.9405	-6.9193	-13.7548
2	0.1300	-1.9945	-1.3927	-0.6599	-0.2186	-0.3024	-0.9682	1.8140	-0.0110	-0.9813
3	-1.9424	-2.3690	-0.4745	-0.5938	-1.6747	-1.5527	0.0527	0.2276	0.4315	-0.4031
4	-2.4353	-0.3924	-0.2406	-0.0685	-0.1756	-0.5804	-0.3610	-0.0662	-0.1040	-0.5393
5	-4.6703	1.3629	-1.2179	-0.2771	-1.8217	0.5362	-1.5284	-0.4245	-0.2020	-0.3820

Fig. 1. Matrix learned after 250 learning cycles. Rows represent different system states and each column represents a blended controller.

Rotor Fault	Original Quadcopter(cm)	RLC Quadcopter (cm)	μ	Improvement
4%	532.81	131.39	100	75.34%
6%	2188.1	650.92	200	70.25%
7%	5439.9	1995.1	250	63.32%

Table 1. Empirical Analysis of RLC improvment in path deviation error against the original controller.

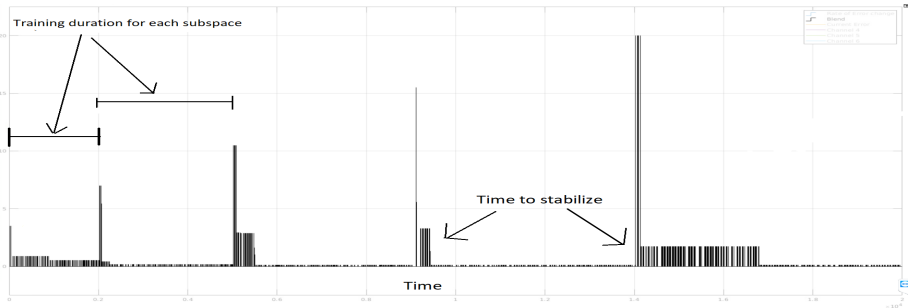


Fig. 2. Evaluation phase τ timings.

5 Reinforcement Learning Blended Control

This section describes the mapping of the RL approach to our experimental domain, i.e. what the RL tuple $\langle S, U, T, R, \gamma \rangle$ means for our quadcopter domain. For the experimental set-up we used an opensource Matlab implementation of a quadcopter flight simulation [20]. This implementation used 4 tuned PID (Proportional Integral Derivative) controllers to control Roll, Pitch, Yaw and Altitude (Figure 3 Right) respectively.

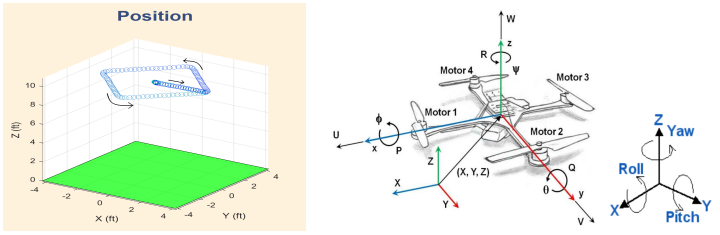


Fig. 3. Left: Simulation Path. Right: Roll , Pitch and Yaw axis of the quadcopter.

Our objective with this simulation is to implement on-line reinforcement learning for faults of unknown origin and magnitude. As far as the authors are aware not much work has been done using RL to learn a blended controller for unknown faults on-line. The simulations core task is trajectory following which is defined as a list of time indexed way point locations in the form of a triplet $(\mathcal{X}, \mathcal{Y}, t)$, presenting desired X and Y position at time t. Given the quadcopters current position at time t as $(\bar{x}, \bar{y})$ the deviation from the trajectory can be computed as:

$$\delta_t = \sqrt{(X - \bar{x})^2 + (Y - \bar{y})^2}$$

The trajectory used is a square and can be seen in Figure 3 Left. We define a single learning cycle for RL as a single execution of the shown trajectory with one unknown faults. The faults are injected at the same point in the trajectory for consistency. After every learning cycle we allow for time to fully stabilize along its trajectory so no residual effects from previous learning cycles will influence future learning quality. Since we have a cyclic trajectory we can continuously repeat the learning cycles and analyse the performance improvements over time and *on-line*. Each learning cycle can be defined in terms of four phases, which are:

1. Stabilize - Ensure nominal operating conditions.
2. Learning - Random fault in specified range and stabilizing using RL.
3. Stabilize - Ensure nominal operating conditions.
4. Evaluate - Test against a benchmark error to check improvement.

The **Stabilize** phases simply ensure the quadcopter returns to its trajectory with acceptable deviation rate and in nominal operating conditions before proceeding to the following phases. The **Learning** phase consists of random fault injection and stabilizing with RL. The **Evaluation** evaluates the current learned policy against a benchmark fault to record the improvements learned over time, Figure 2.

RL States The space of unknown faults is simply too large to apply learning to each fault individually. We hence classify the states S for our RL implementation in terms of the *effect* that unknown faults have on the trajectory following task. For this metric we chose the deviation rate from the desired trajectory, which we define as:

$$\rho = \frac{d}{dt} \sum_{i=0}^{4} \delta_{t-i},$$

where δ_i is the trajectory error between current and desired position at time step $t - i$. We then define S as a partition over the parameter space of ρ into N regions of equal size.

RL Actions Our action space U is defined in terms of blended control for which we use a linear combination of predefined controllers. In other words, given a set of M predefined controllers $\Lambda = \{\Lambda_1, \cdots, \Lambda_M\}$ and corresponding weights $\{\varphi_1, \cdots, \varphi_M\}$, our applied blended control is given by

$$\Lambda^* = \sum_{i=1}^{M} \varphi_i \Lambda_i. \tag{1}$$

where $\sum_{i=1}^{M} \varphi_i = 1$. We define our action space U as a partition of size P over the parameter space of φ. The granularity of this partition will dictate how many different blended controllers are being learned on.

We then define our Q-Matrix as $Q(S, U)$ and set N and P to 5 and 10 respectively. In other words we are learning the optimal blended controller for each deviation-rate sub-partition. The size of the matrix increases drastically with the size of the partitions. We hence chose small enough partition sizes to concentrate the learning into a smaller region. The transition function T maps the current state and action chosen to the new state. In our context T is already given by the simulation itself.

For RL to be able to learn *on-line* we must have some performance metric to evaluate how well a controller performed during the fault recovery for each separate learning cycle. For this metric we choose the time, τ, until the quadcopter is stabilized on its desired trajectory.

RL Rewards Since we use τ as our performance metric we must also compute a baseline value to compare this against, which indicates positive or negative reward for the blended controller used. Since we have no prior knowledge about the faults, we compute $\bar{\tau}$, a running average of τ. For each subspace of S^i we define $\bar{\tau}^i$ to allow larger errors more time to stabilize. We assign credit based on the performance against the running average.

$$\mathcal{C} = \begin{cases} 1 \text{ if } \tau \leq \bar{\tau}^i \\ \text{-1 if } \tau > \bar{\tau}^i \end{cases}$$

given that $max(\rho) < S^i$. The $\bar{\tau}$ values for each fault mode after the RL simulation can be found in Table 2.

S^i	S^1	S^2	S^3	S^4	S^5
$\bar{\tau}^i$ (sec)	2.8	3.6	3.9	4.3	4.7

Table 2. $\bar{\tau}^i$ values for each partition of S after the RL simulation.

Credit Assignment Problem The *Credit Assignment Problem* [21] refers to the problem of identifying **which** action was the one deserving credit. In our case the problem is identifying which partition S^i should receive the credit after a learning phase. This is because ρ will naturally vary across multiple sub-partitions of S during a single learning cycle. We hence apply RL for each of the sub-partitions. The blended controller used to control the system is changed when ρ transitions from its current S^i to S^{i-1} or S^{i+1}. Figure 4 shows various signals and the bounds of each S^i indicated as *Levels* from the quadcopter simulation during a single learning cycle. The Green signal represents the changing ρ. We indicate the transition between sub-partitions of S with red arrows. Notice the blue signal representing the applied control signal changes when the deviation magnitude changes. However this implementation makes it difficult to assign credit to a single subspace, since learning was potentially applied on multiple subspaces. We combat this by assigning proportional credit to each subspace depending on how long the value of ρ stayed in each of the parameter subspaces. That is, the credit assigned to each S^i is:

$$\mathcal{R}(S^i) = \mathcal{C} * \frac{t}{\tau}$$

where t is the duration that ρ was in the bounds of S^i and $\mathcal{C}$ and τ are as previously defined.

For this implementation we do not use γ, the future expected reward. Estimating the expected deviation rate ρ_{t+1} is challenging as the controller is synthesised on-line and the faults are unknown. In future work we will extend the reward function to include the γ term.

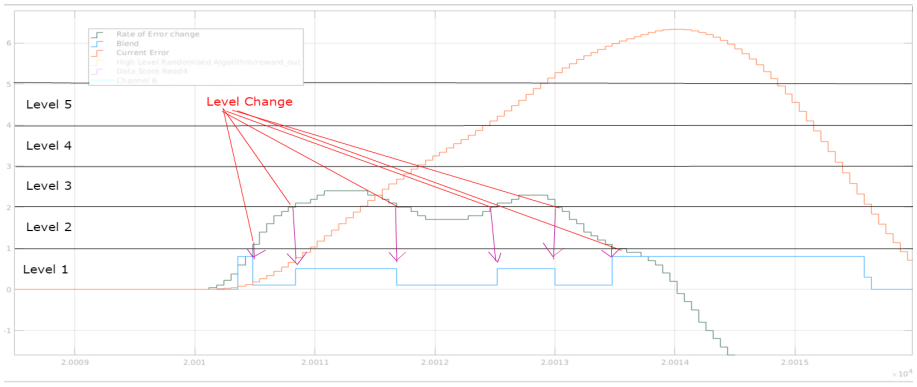


Fig. 4. Graph showing ρ (green), current error (orange) and blended controller used (blue). The subspaces $S^1, S^2, ..., S^5$ are indicated along the Y-axis using the *Level* terminology.

Error Generation We focus solely on unknown rotor faults for this article for simplicity but this framework works for any unknown faults that cause trajectory deviations. More specifically, we use a multiplicative term ι on the rotor speed to represent the unknown fault. To achieve an even distribution of learning we randomize ι, such $0 \leq \iota \leq \Gamma$, where Γ is the upper limit on the fault. We incrementally increase Γ after a number of learning cycles. This will give a larger variance of deviations allowing the system to apply learning across all sub-partitions. Figure 5 shows the rotor fault magnitudes used over 250 learning cycles. Note this figure shows the benchmark errors used as well as randomized errors for training.

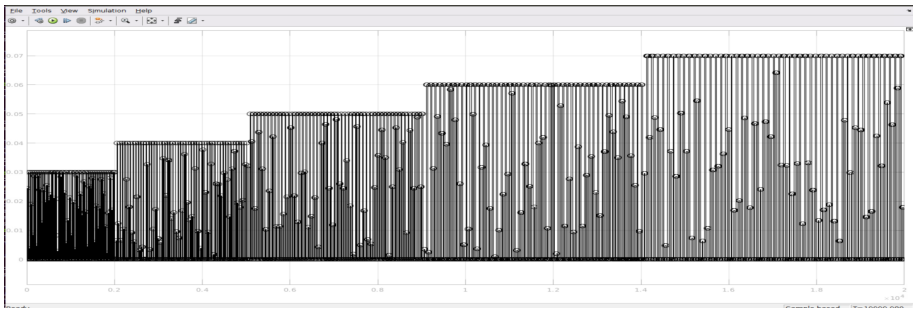


Fig. 5. Rotor Fault magnitude ι and benchmark errors Γ throughout simulation. Note: larger errors are given longer time to stabilize.

5.1 Simulation Details

For completeness we will give a full list of low level controllers and other parameters used for the experiments. The PID controller coefficients for nominal and fault (indicated by superscript N and F) controller for each control axis (indicated by subscript ϕ, θ, ψ) can be seen in Table 3. It is worth noting that these controllers are not tuned for any specific fault; they are simply tuned with a more "aggressive" coefficient. Figure 1 shows the matrix after 250 learning cycles. We set the initial value for Γ to 3% and increased this after every 50 learning iterations up to 7%. We apply the fault to the same rotor every time.

Controller	λ_ϕ^N	λ_ϕ^F	λ_θ^N	λ_θ^F	λ_ψ^N	λ_ψ^F
P	2	10	2	10	4	14
I	1.1	5	1.1	5	0.5	2
D	1.2	3	1.2	3	3.5	5

Table 3. Low level nominal and fault controller tuning for each control axis.

6 Conclusion

In this article we presented a novel reinforcement learning approach for online, real-time learning for unknown faults. We empirically demonstrated the effectiveness of this approach on a Quadcopter trajectory following task. We noticed a 63-75% decrease in the trajectory deviation due to an unknown fault after a small number of learning cycles. In this set of experiments only rotor faults where investigated but given appropriate controller pairs for blending the authors believe this approach will work for other system disturbances such as wind or sensor faults. Since we are learning on the disturbance space ,instead of the fault parameter space, this learning approach will work for most faults that cause a path deviation. The novelty of this approach is that the learning phase can be conducted on-line and needs very few iterations before converging compared to traditional RL methods. Further more, we are not aware of any other RL based approach for FTC under novel faults. This could provide adaptive FTC capabilities to systems that are not easily fixable or reconfigurable such as satellites or space rovers. Our future work includes focusing on improving the learning process and attempting training on a multitude of errors. For simplicity, this article also only explores a two controller blending strategy but in theory this is not a limitation. Larger sets of predefined controllers for blending can be trained using our described method but with a considerably larger number of training phases which will also be explored in future work.

References

1. Richard S Sutton, Andrew G Barto, and Ronald J Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, 12(2):19–22, 1992.
2. Angela P Schoellig, Fabian L Mueller, and Raffaello DAndrea. Optimization-based iterative learning for precise quadrocopter trajectory tracking. *Autonomous Robots*, 33(1-2):103–127, 2012.
3. Fabian L Mueller, Angela P Schoellig, and Raffaello D'Andrea. Iterative learning of feed-forward corrections for high-performance tracking. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 3276–3281. IEEE, 2012.
4. Markus Hehn and Raffaello D'Andrea. A frequency domain iterative feed-forward learning scheme for high performance periodic quadrocopter maneuvers. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 2445–2451. IEEE, 2013.
5. Jemin Hwangbo, Inkyu Sa, Roland Siegwart, and Marco Hutter. Control of a quadrotor with reinforcement learning. *IEEE Robotics and Automation Letters*, 2(4):2096–2103, 2017.
6. Michael C. Koval Michael L. Littman, Christopher R. Mansley. "Autonomous quadrotor control with reinforcement learning".
7. Mogens Blanke, Michel Kinnaert, Jan Lunze, Marcel Staroswiecki, and J Schröder. *Diagnosis and fault-tolerant control*, volume 691. Springer, 2006.
8. Ron J Patton. Fault-tolerant control. *Encyclopedia of systems and control*, pages 422–428, 2015.
9. Xiang Yu and Jin Jiang. A survey of fault-tolerant controllers based on safety-related issues. *Annual Reviews in Control*, 39:46–57, 2015.
10. Matthew Kuipers and Petros Ioannou. Multiple model adaptive control with mixing. *IEEE Transactions on Automatic Control*, 55(8):1822–1836, 2010.
11. Youmin Zhang and Jin Jiang. "Integrated active fault-tolerant control using IMM approach". *Transactions on Aerospace and Electronic Systems*, 37(4):1221–1235, 2001.
12. Jin Jiang and Xiang Yu. "Fault-tolerant control systems: A comparative study between active and passive approaches". *Annual Reviews in Control*, 36:60–72, 2012.
13. Kemal Büyükkabasakal, Bariş Fidan, and Aydogan Savran. Mixing adaptive fault tolerant control of quadrotor UAV. *Asian Journal of Control*, 19(5):1–14, 2017.
14. Jan Lunze. From fault diagnosis to reconfigurable control: A unified concept. In *Control and Fault-Tolerant Systems (SysTol), 2016 3rd Conference on*, pages 413–421. IEEE, 2016.
15. Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.
16. Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
17. Kumpati S Narendra and Zhuo Han. The changing face of adaptive control: the use of multiple models. *Annual Reviews in Control*, 35(1):1–12, 2011.
18. Said G Khan, Guido Herrmann, Frank L Lewis, Tony Pipe, and Chris Melhuish. Reinforcement learning and optimal adaptive control: An overview and implementation examples. *Annual Reviews in Control*, 36(1):42–59, 2012.
19. Weicun Zhang. Stable weighted multiple model adaptive control: discrete-time stochastic plant. *International Journal of Adaptive Control and Signal Processing*, 27(7):562–581, 2013.

20. D. Hartman, K. Landis, M. Mehrer, S. Moreno, and J. Kim. *Quadcopter Simulation*.
21. RICHARD S. SUTTON. *TEMPORAL CREDIT ASSIGNMENT IN REINFORCEMENT LEARNING*. PhD thesis, 1984. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-05-11.