UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA QUÍMICA

RODRIGO FONTES MANTOVANI

DEEP REINFORCEMENT LEARNING FRAMEWORK FOR FAULT DETECTION IN CONTINUOUS CHEMICAL PROCESSES

BELO HORIZONTE - MG
2024

Rodrigo Fontes Mantovani

Deep reinforcement learning framework for fault detection in continuous chemical processes

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Química da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Engenharia Química.

Área de concentração: Engenharia de Sistemas em Processos

Orientador: Éder Domingos de Oliveira

Coorientador: Gustavo Matheus de Almeida

Belo Horizonte - MG
2024

Mantovani, Rodrigo Fontes.
M293d Deep reinforcement learning framework for fault detection in continuous chemical processes [recurso eletrônico] / Rodrigo Fontes Mantovani. – 2024.
1 recurso online (83 f. : il., color.) : pdf.

Orientador: Éder Domingos de Oliveira.
Coorientador: Gustavo Matheus de Almeida.

Dissertação (mestrado) – Universidade Federal de Minas Gerais, Escola de Engenharia.

Apêndices: f. 76-83.

Bibliografia: f. 72-75.

1. Engenharia química – Teses. 2. Processos de fabricação – Teses. 3. Aprendizado por reforço – Teses. 4. Aprendizado do computador – Teses. 5. Framework (Programa de computador) – Teses. 6. Detecção de anomalias (Computação) – Teses. I. Oliveira, Éder Domingos de. II. Almeida, Gustavo Matheus de. III. Universidade Federal de Minas Gerais. Escola de Engenharia. IV. Título.

CDU: 66.0(043)

Ficha catalográfica elaborada pelo bibliotecário Marcio Anderson de Andrade Gomes CRB/6 2812
Biblioteca Prof. Mário Werneck, Escola de Engenharia da UFMG

UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA QUÍMICA

FOLHA DE APROVAÇÃO

"DEEP REINFORCEMENT LEARNING FRAMEWORK FOR FAULT DETECTION IN CONTINUOUS CHEMICAL PROCESSES"

Rodrigo Fontes Mantovani

Dissertação submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Química da Escola de Engenharia da Universidade Federal de Minas Gerais, como parte dos requisitos à obtenção do título de **MESTRE EM ENGENHARIA QUÍMICA**.

314ª DISSERTAÇÃO APROVADA EM 26 DE MARÇO DE 2024 POR:

Documento assinado eletronicamente por **Gustavo Matheus de Almeida**, **Usuário Externo**, em 02/07/2024, às 15:35, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Roberto da Costa Quinino**, **Coordenador(a) de curso**, em 02/07/2024, às 17:04, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Antonio de Padua Braga**, **Membro**, em 04/07/2024, às 08:40, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Eder Domingos de Oliveira**, **Professor do Magistério Superior**, em 24/07/2024, às 17:30, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3096869** e o código CRC **CBE8C63E**.

Referência: Processo nº 23072.214445/2024-11 SEI nº 3096869

To my mother, my father and my sisters

AGRADECIMENTOS

Agradeço à Universidade Federal de Minas Gerais (UFMG) por ser crucial na minha formação enquanto ser humano. Enquanto discente, aprendi que a universidade pública vai muito além da sala de aula, e não posso deixar de expressar a minha gratidão e admiração por essa instituição que tanto contribui para o Brasil e para o mundo. Vida longa à educação pública, gratuita e de qualidade.

Agradeço ao meu orientador e professor, Éder Domingos de Oliveira, pela disponibilidade e prestatividade. Aos professores e membros da banca, Roberto da Costa Quinino e Antônio de Pádua Braga, agradeço pelos ensinamentos e por contribuírem imensamente com o presente trabalho e com a minha trajetória na pós-graduação.

Ao meu coorientador e professor, Gustavo Matheus de Almeida, agradeço por ter me apresentado uma nova forma de enxergar a minha vida profissional, através do trabalho de conclusão de curso, da iniciação científica e da pós-graduação.

Aos mestres do Departamento de Engenharia Química (DEQ) e do Instituto de Ciências Exatas (ICEx), agradeço pelo conhecimento compartilhado.

Aos meus pais, Elaine e Everardo, por serem meu maior alicerce e fonte de inspiração. Com vocês, aprendi o sentido que hoje guia a minha vida. Por vocês, prometo sempre buscar o melhor que posso ser, repassando os ensinamentos que com vocês aprendi. Minha gratidão e meu amor não cabem e jamais caberão em palavras.

A Flávia e Lívia, por serem minhas primeiras amigas, minhas irmãs, e por todo o amor, cuidado, carinho e apoio que tive e com certeza terei por toda a minha vida.

Às minhas sobrinhas, Olivia e Estela, por darem um novo significado ao amor e à vida. Ao meu cunhado Shin, pela parceria, suporte e carinho.

Aos meus amigos, que me mostram dia após dia que podemos, com certeza, escolher e cultivar laços familiares que vão além do sangue.

Às famílias Chartuni Mantovani e Oliveira Fontes, pelo amor que sempre me foi dado desde quando era muito novo para entendê-lo.

A Carolina, por me motivar a ser melhor em todos os sentidos da palavra, por me fornecer apoio fundamental em todos os momentos de dificuldade e por compartilhar comigo os momentos mais felizes. Sua felicidade é a minha.

“A soul in tension that’s learning to fly
Condition grounded, but determined to try”

David Jon Gilmour

RESUMO

Processos industriais modernos estão sujeitos a padrões cada vez mais altos de qualidade, segurança, sustentabilidade e economia. A implementação de controle em malha fechada visa assegurar o cumprimento desses padrões, compensando perturbações e mudanças no processo. Contudo, falhas, ou seja, desvios não permitidos em propriedades ou variáveis do sistema, ainda são comuns e podem afetar significativamente o processo, dificultando a aderência a esses requisitos. Assim, práticas de monitoramento de processos para detectar, diagnosticar e corrigir falhas tornam-se cada vez mais cruciais. Em um contexto de Aprendizado de Máquina, boa parte das abordagens tradicionais se baseiam em aprendizado supervisionado, com o objetivo de determinar o mapeamento entre os dados do processo e algum conjunto discreto de classes, neste caso os tipos de falhas. Recentemente, abordagens de detecção de falhas industriais baseadas em Aprendizado por Reforço (Reinforcement Learning, RL) se fazem cada vez mais presentes na literatura. Os métodos de RL consistem no desenvolvimento de agentes inteligentes que aprendem a interagir com determinado ambiente para maximizar recompensas. No entanto, a maioria das abordagens atuais foca em sistemas simples com poucas variáveis. Este projeto visou criar um framework de RL para detecção de falhas em processos químicos contínuos, que envolvem múltiplas variáveis e dinâmicas complexas. Por meio do algoritmo Deep Q-Learning, o framework proposto foi capaz de detectar vinte falhas diferentes no Tennessee Eastman Process (TEP), benchmark amplamente utilizado na comunidade científica que consiste em uma simulação realista de um processo químico contínuo. Os modelos superaram um modelo baseline baseado em Análise de Componentes Principais (PCA) em métricas como taxas de falso alarme e detecções perdidas, bem como tempo até detecção, para a maioria das falhas. Notavelmente, para falhas caracterizadas por alta dificuldade de detecção, obteve-se taxa média de detecções perdidas de 12,49%, comparadas a uma taxa mínima de 79,59% para o PCA.

Palavras-chave: detecção de falhas; processos químicos contínuos; aprendizado por reforço; *deep q-learning*; aprendizado de máquina.

ABSTRACT

Modern industrial processes are subject to increasingly higher quality, safety, environmental and economical standards. The implementation of closed-loop control aims to guarantee that these standards are met by compensating the effects of disturbances and changes that might occur in the process. However, faults, i.e., unpermitted deviations of at least one characteristic property or variable of the system, can still happen in most industrial processes and have an impact on such requirements. Therefore, process monitoring tasks that detect, diagnose and remove faults are increasingly more necessary. In the field of Machine Learning, some of the most relevant approaches to fault detection are supervised learning-based methods, in which the learning process focuses on determining the mapping between input variables, i.e., process data, and some discrete set of classes, in this case the fault types. Recently, there has been an increase in industrial fault detection approaches based on Reinforcement Learning (RL), which consists on building intelligent agents that interact with certain environment by performing actions and receive rewards based on the quality of the actions performed. However, the majority of the methods currently present in literature focuses in equipment such as industrial rotating machinery and rely on datasets with a small number of variables. Hence, this project's objective is to develop a reinforcement learning framework to perform fault detection in continuous chemical process, which have a large number of variables and complex dynamics. The proposed framework was able to train, validate and test RL agents with the Deep Q-Learning algorithm and use these agents to detect twenty different fault types in the simulated Tennessee Eastman Process benchmark. The model reached higher performance according to three selected metrics (false alarme rate, missed detection rate and time to detection) for most faults if compared to a Principal Component Analysis baseline model. Notably, for faults known to be hard-to-detect, the proposed framework reached mean missed detection rates of 12,49%, as compared to a minimum of 79,59% for the PCA baseline approach.

Keywords: fault detection; continuous chemical processes; reinforcement learning; deep q-learning; machine learning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIMBOLOS

$p(a\|b)$	Conditional probability of a, given b	[ ]
w_{ji}	Weight relating the j^{th} node of the previous layer with the i^{th} node of the current layer in an artificial neural network	[ ]
α	Learning rate	[ ]
L	Loss function	[ ]
J	Cost function	[ ]
S_t	State of an environment during time step t	[ ]
A_t	Action performed during time step t	[ ]
R_t	Reward received during time step t	[ ]
$\mathcal{S}$	Set of all possible states	[ ]
$\mathcal{A}$	Set of all possible actions	[ ]
$\mathcal{R}$	Set of all possible rewards	[ ]
s	A given generic state	[ ]
r	A given generic reward	[ ]
a	A given generic action	[ ]
π	Reinforcement learning policy	[ ]
γ	Discount factor	[ ]
G_t	Expected future rewards	[ ]
$v_\pi(s)$	State-value function of state s	[ ]
$q_\pi(s,a)$	Action-value function of action a in state s	[ ]
$\mathbb{E}_\pi$	Expected value of any variable while under a policy π	[ ]
ϵ	Probability of selecting a random action under ϵ-greedy policy	[ ]
V	Estimated state-value for Monte Carlo methods	[ ]
Q	Estimated action-value for Monte Carlo methods	[ ]
θ	Set of parameters in an artificial neural network	[ ]

SUMÁRIO

1. INTRODUCTION

Due to ever-expanding technological advances, the usage of data to support and enhance the capability of decision-making processes has been growing exponentially and becoming more necessary than ever. The definition of Big Data arises from this phenomenon and refers to structured or unstructured data that became too large for the traditional data-processing softwares to handle, due to ever-increasing volume, velocity and variety. Therefore, the development of new technologies that facilitate data analysis, storage and management is a pivotal need for most fields, such as industry, marketing, medicine, research and development, etc. With more data, there is the opportunity of gathering more information, but also the increasing volume provides implicit difficulties and complexity for extracting and using this information to achieve many goals (HARRIS et al., 2023; PEDOTA, 2023).

For the process industry, with the transformation of the traditional processes into smart ones, supported by phenomena such as **Big Data** and **Industry 4.0**, most processes are equipped with several sensors, which implies the increasing gathering of process-related data, which can be used for many purposes, such as process control, knowledge discovery, process understanding, decision-making, production yield prediction and maximization, process optimization and **fault detection and diagnosis** (FDD). The latter, which is related to the process monitoring field, aims to detect, identify and eliminate undesired deviations in processes to ensure that the increasingly more demanding quality, safety and economical requirements of current industrial processes are met (ISERMANN; BALLÉ, 1997; PARK; FAN; HSU, 2020).

FDD techniques are significantly critical for maintaining desirable performance in continuous chemical processes and achieving high process yields and throughput, and many different methods are applied. (PARK; FAN; HSU, 2020). In the field of **Machine Learning** (ML), which refers to methods capable of detecting patterns from datasets and using these patterns to make decisions, predict future data and classify different groups (MURPHY, 2012), many methods can be used for this purpose. Usually, the ML branch known as **supervised learning** (SL) is the most used approach for FDD methods, and it consists in

designing algorithms capable of learning the mapping between process data and different fault types.

The ML branch known as **reinforcement learning** (RL) refers to the construction of agents that can learn to perform desired tasks by interacting with a dynamic environment and receiving rewards that qualify its behavior. Some recent approaches aim to design RL agents to perform FDD in industrial processes, in order to address limitations of traditional SL methods and improve performance in specific contexts (BELHADI et al., 2021; DING et al., 2019; MALIK; SHARMA; MISHRA, 2020; QIAN; LIU, 2022; WANG; XUAN, 2021; WU et al., 2022, 2022; ZHONG; ZHANG; BAN, 2023). However, most of these methods refer to equipment such as rotating machinery and rolling bearers, that generate mostly univariate datasets and a maximum of eight variables. Since continuous chemical processes have several unit operations and highly complex and non-linear relations between variables, these approaches do not explicitly translate to FDD in such processes. Therefore, the current work aims to design a RL framework to perform fault detection in continuous chemical processes. The structure will be further divided into 6 sections:

- **Section 2** contains the general and specific objectives of this dissertation proposal;
- **Section 3** refers to a literature review in 5 parts: (1) fault detection in continuous chemical processes; (2) machine learning; (3) reinforcement learning; (4) usual reinforcement learning applications; (5) applications of reinforcement learning for fault detection and diagnosis in continuous chemical processes;
- **Section 4** provides a case study of the Tennessee Eastman Process, a commonly used benchmark process simulation which will be the basis for the fault detection framework;
- **Section 5** addresses the proposed methodology;
- **Section 6** contains results and discussion of the proposed framework for twenty different fault types in relation to their degree of detection difficulty;
- **Section 7** contains conclusions regarding the dissertation.

2. OBJECTIVES

2.1. GENERAL OBJECTIVES

The objective of the current work is to develop a deep reinforcement learning framework that creates an agent capable of performing fault detection in multivariate, continuous chemical processes.

2.2. SPECIFIC OBJECTIVES

- To build a custom reinforcement learning environment that adapts the fault detection task, usually solved by supervised learning approaches;
- To define the agent-environment interaction that makes it possible to use such framework;
- To train, validate and test reinforcement learning agents with two neural network architectures of varying complexity, analyzing performance metrics such as false alarm rate, missed detection rate and time to detection;
- To compare the reinforcement learning methods developed to baseline methods found in literature.

3. LITERATURE REVIEW

3.1. FAULT DETECTION IN CONTINOUS CHEMICAL PROCESSES

Modern industrial processes are subject to increasingly higher quality, safety, environmental and economical standards. The implementation of closed-loop control aims to guarantee that these standards are met, by compensating the effects of disturbances and changes that might occur in the process. However, **faults**, defined by Iserman and Ballé (1997) as unpermitted deviations of at least one characteristic property or variable of the system, can still happen in most industrial processes.

There are several types of faults, including **process parameter changes**, such as catalyst poisoning and heat exchanger fouling, **disturbance parameter changes**, e.g., an extreme change in a process feed stream concentration or temperature, and also problems in **actuators** and **sensors**. The occurrence of faults can significantly impact the process and make it more difficult to meet several requirements, and, therefore, **process monitoring** tasks that aim to detect, diagnose and remove the disturbances that might cause faults are increasingly more necessary (CHIANG; RUSSELL; BRAATZ, 2000; KESAVAN; LEE, 1997).

According to Chiang et al. (2000), the four tasks associated with process monitoring are: **fault detection**, i.e., determining if a fault occurred; **fault identification**, i.e., identifying the variables that are most relevant to diagnosing the cause of the fault; **fault diagnosis**, which means determining which fault is responsible for the problem; and **process recovery**, representing the intervention necessary to remove the effects of the fault. The combination of these tasks defines a process monitoring loop, as shown in **Figure 1**. Frequently, methods that combine detection and diagnosis stages are referred to as **Fault Detection and Diagnosis (FDD)** methods.

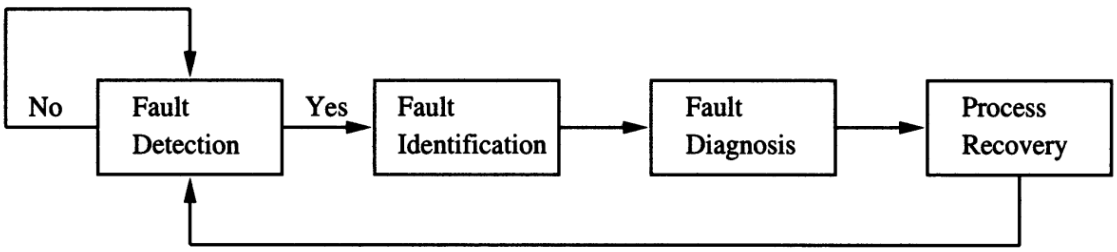


Figure 1: Schema of process monitoring loop (CHIANG; RUSSELL; BRAATZ, 2000).

Chiang et al. (2000) also classify process monitoring methods into three main approaches. The first approach is the one of **data-driven methods**, derived

directly from process data. In large-scale industrial systems, the amount of data produced is increasingly expanding, which makes relevant information more available, however making it more difficult for humans to analyze and extract it. The data-driven techniques are highly dependent on the quality and quantity of historical data, and several methods belong to these categories, among them the Shewhart, CUSUM and EWMA control charts, Principal Component Analysis (PCA), Partial Least Squares (PLS), Fisher Discriminant Analysis (FDA) and Canonical Variate Analysis (CVA), to name a few.

The second approach refers to **analytical methods**, that rely on detailed mathematical models for obtaining features such as residuals, parameter estimates and state estimates. In these methods, fault detection and diagnosis occurs by comparing observed features with the ones associated with normal operation. In complex, non-linear and large-scale systems, such as continuous processes, many resources might be required to obtain detailed mathematical models, making it impractical or, at least temporarily, unfeasible.

The third approach represents **knowledge-based methods**, which rely on qualitative models that can be obtained from causal analysis, expert systems and pattern recognition (CHIANG; RUSSELL; BRAATZ, 2000). In causal analysis, qualitative and semi-quantitative relations between faults and symptoms are gathered, relying on first principles. Expert systems are used to simulate reasoning from human experts, combining first principles and structural descriptions of the system to diagnose faults. Pattern recognition techniques aim to obtain and identify data patterns and its relation with specific faults, without the need to model the process structure. Popular examples are artificial neural networks (ANN) and self-organizing maps (CHIANG; RUSSELL; BRAATZ, 2000).

3.2. MACHINE LEARNING

The search for methods that are capable of performing tasks usually exclusive to human beings has been developed for decades, combining concepts from statistics, artificial intelligence, philosophy, information theory, biology, cognitive science, computational complexity, control theory, among others. For computer algorithms and methods, it is defined that a computer program learns from experience E, with respect to a class of tasks T and performance measurement P, if its performance in task T, measured by P, improves with experience E (MITCHELL, 1997).

Murphy (2012) defines **machine learning** as a set of methods that are capable of detecting patterns in datasets, as well as utilizing those patterns to make decisions and predict future data. There are basically three main types of

machine learning problems, known as (1) **supervised learning**, (2) **unsupervised learning** and (3) **reinforcement learning**. In **supervised learning**, the main objective is predictive, i.e., to learn the mapping between inputs x and outputs y based on a labeled dataset containing input-output pairs, the **training set**. The output variable can be categorical, as in classification problems (e.g., classifying different operating modes in a chemical process), or real-valued, i.e., in regression problems (e.g., predicting the superheated steam output in boilers) (MURPHY, 2012).

In **unsupervised learning** problems, the dataset is unlabeled, i.e., there is no formalization regarding the desired outputs of the model in terms of its inputs. Thus, the objective is to obtain models in the form $p(x_i|\theta)$, instead of typical supervised learning models in the form $p(y_i|x_i, \theta)$. These methods are also known as **knowledge discovery methods**, and its problems are defined by the search for interesting data patterns, as well as the interpretation of those patterns and the formulation of strategies that allow the utilization of these identified structures (MURPHY, 2012).

Sutton and Barto (2018) define **reinforcement learning** (RL) as the third paradigm in machine learning, providing a different problem structure. In this paradigm, the objective is to construct an agent that is capable of learning to follow some specific behavior by multiple trial-and-error interactions with a dynamic environment. After the agent selects an action, it receives a reward that qualifies its behavior, without receiving, however, the information of what would be the best action possible. Therefore, the agent must collect experience regarding possible actions, states, transitions and rewards and, with that, learn an optimal behavior, i.e., a mapping of the actions that maximize the agent's rewards (KAELBLING; LITTMAN; MOORE, 1996).

Some works might also present a fourth class of machine learning methods, named **semi-supervised learning**, that use datasets with both labeled and unlabeled data, a very common characteristic in real-word applications. Those methods combine appropriate techniques of both supervised and unsupervised learning in order to complete missing information and to allow the unlabeled part of the dataset to be used for tasks typically applied only to labeled data (PRAVEEN KUMAR; AMGOTH; ANNAVARAPU, 2019). **Figure 2** shows a taxonomy diagram of machine learning and its methods.

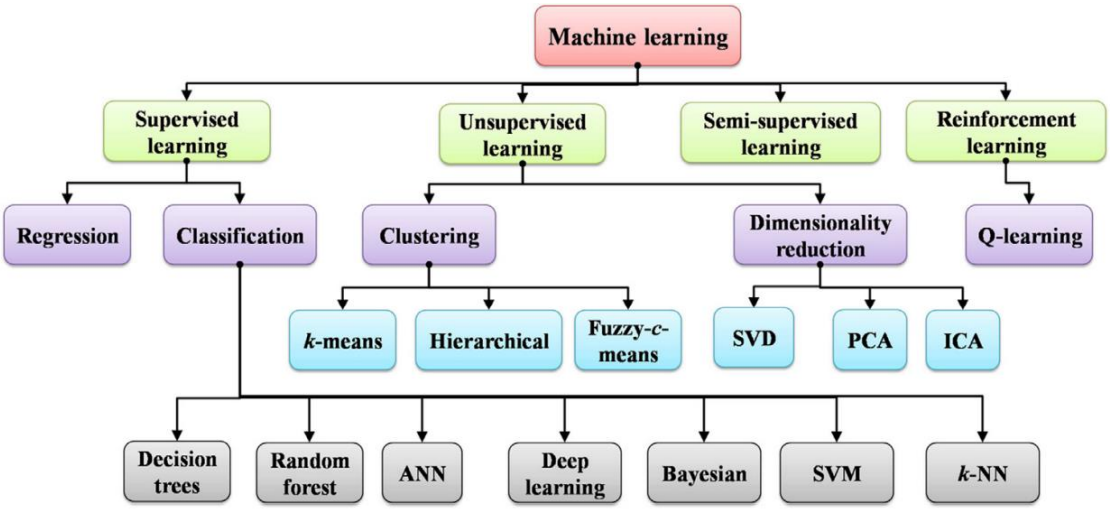


Figure 2: Taxonomy of machine learning techniques. (PRAVEEN KUMAR; AMGOTH; ANNAVARAPU, 2019)

Between the techniques related in Figure 2, two will be briefly discussed in the next topics for contextualization purposes, as they will be used in the development of this work. They are **Artificial Neural Networks** (ANN) and **Principal Component Analysis** (PCA).

3.2.1. Artificial neural networks

Artificial neural networks (ANNs), or simply neural networks, are machine learning methods developed based on the architecture of real biological neurons, as the ones found in the human brain. They consist of parallel distributed systems composed of simple processing units (nodes) and are used to compute mostly nonlinear mathematical functions. The capacity for learning through examples and for generalizing the learned information is the main reason for using ANNs to solve problems. (BRAGA; LUDERMIR; CARVALHO, 2000). The three main components of ANNs are (1) the neurons (or nodes), (2) the network topology and (3) the learning rules. Each **neuron** receives multiple inputs from weighted connections (similar to the synaptic strength of biological neurons). If the weighted sum of inputs exceeds a pre-defined threshold, the neuron is activated and processes the received signal through a **transfer function** (or **activation function**), sending the resulting output to the following neurons (ZOU; HAN; SO, 2009). **Figure 3** shows an artificial neuron model.

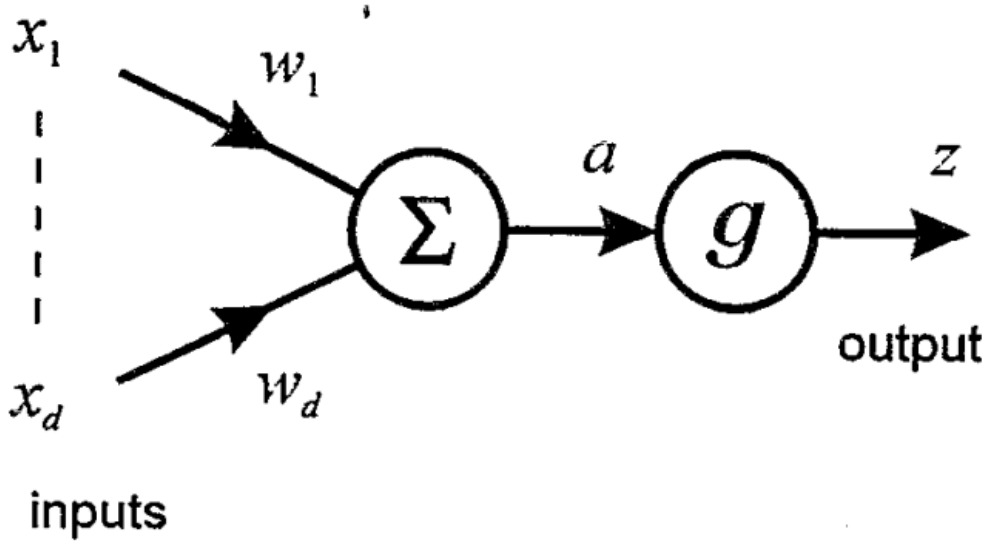


Figure 3: A single artificial neuron model. The model receives an input vector x, performs a sum weighted by w, defined as a, that goes through an activation function g and returns the output z (BISHOP, 1994).

The nodes of a neural network are organized in a **layer topology**. Generally, ANNs have **input layers**, i.e., responsible for receiving inputs, **output layers**, i.e., that produce the final outputs, and **hidden layers**, which are intermediate layers between the inputs and the outputs. The connections between the neurons of the ANN are from two general types, the first being direct input-output connections, in which the signal processed by each node always goes in the direction of the following layers. The second type involves connections by which the output of a neuron can return to neurons from previous layers or the same one. According to the respective connection types, neural networks are classified as **feedforward networks** and **feedback networks**. In **Figure 4,** three general examples of ANN topologies are represented (ZOU; HAN; SO, 2009).

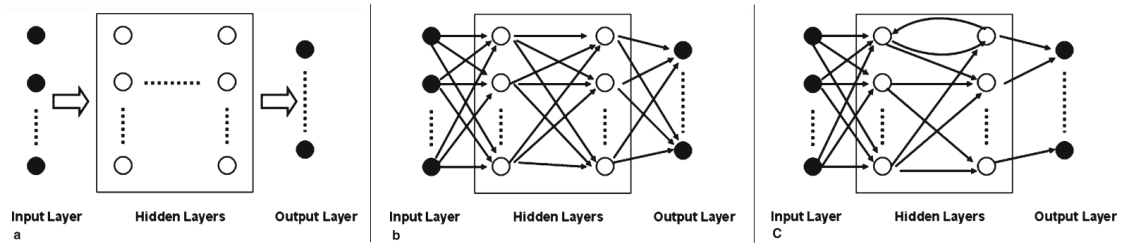


Figure 4: (a) General ANN topology. **(b)** Feedforward ANN (Multilayer perceptron). **(c)** Feedback ANN. Adapted from (ZOU; HAN; SO, 2009)

The third pillar of neural networks refers to **learning**, and any of the three (or four, in case semi-supervised learning is considered) classes of machine learning methods can rely on ANNs. The training is performed by the iterative update of parameters, i.e., the weights associated with the network connectionsm in terms of the desired output values. Some classical examples of neural networks are the Perceptron, the Hopfield Net and the Kohonen maps (ZOU; HAN; SO, 2009).

One of the most common methods used to perform the parameter estimation in a neural network is known as **gradient descent**, in which a loss function is defined, for example, based on the errors between the network output and the corresponding real desired value. The update happens based on the gradient of this loss function in terms of the parameters, and its proportional to a value, usually small, called **learning rate**. The objective is to establish a function whose minimization would make the network perform the task according to its desired objective. The weights of a particular layer will be updated, in each step of the algorithm, according to Equation (1), or with equations derived from it:

$$\Delta w_{ji}^{(\tau)} \equiv w_{ji}^{(\tau)} - w_{ji}^{(\tau-1)} = -\alpha \left. \frac{\partial L}{\partial w_{ji}} \right|_{w^{(\tau)}} \quad (1)$$

in which τ is the iteration, α is the learning rate, L is the loss function and w_{ji} is the weight that relates the j^{th} node of the previous layer with the i^{th} node of this layer (BISHOP, 1994).

The main algorithm used to implement the gradient descent method is called **backpropagation**. In a neural network, the set of all neurons forms the **network function**. This function is generated by propagating the input through all nodes and all respective activation functions. By using differentiable activation functions, e.g., sigmoid and hyperbolic tangent, it is also guaranteed that the function computed by the neural network will be differentiable. The same will apply, accordingly, to the error function of the ANN (ROJAS, 1996).

The algorithm can be divided in two steps. Zhang et al. (2021) defined the backpropagation algorithm for a simple neural network, composed by a two-neuron series, with the associated weights for each neuron, $W^{(1)}$ and $W^{(2)}$, representing all of the parameters that must be learned in order to minimize a cost function $J = L + s$, in which L is the loss function (error) and s is a regularization parameter. It is possible to represent the first step of the algorithm, **feed-forward**, as shown in **Figure 5**.

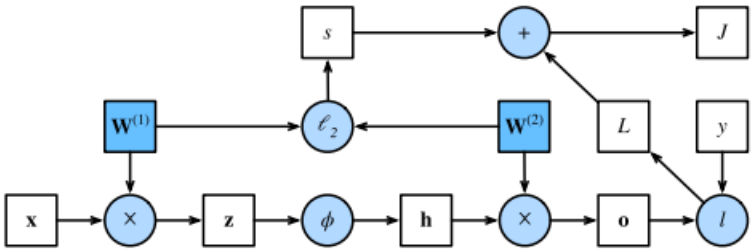


Figure 5: Diagram of the feed-forward step in a simple neural network, composed by a series of two neurons. In the diagram, , $\mathrm{z} = \mathbf{W^{(1)}}\mathrm{x}$, $\mathbf{h} = \phi(\mathrm{z})$, $\mathbf{o} = \mathbf{W^{(2)}h}$, L is the loss function, s is a regularization term and J is the objective function to be minimized (ZHANG et al., 2021)

In Equations (2)-(9), the mathematical formulation that guides the process of obtaining of the derivatives of the objective function in terms of the network parameters, which consists mostly of the chain rule of differential calculus, is demonstrated. The gradients of the objective function in terms of its parameters L and s are as follows.

$$\frac{\partial J}{\partial L} = 1, \qquad \frac{\partial J}{\partial s} = 1 \tag{2}$$

As the ANN produces an output $\mathbf{o}$, it is possible to obtain the gradient of the objective function with respect to $\mathbf{o}$ by the following way, using the chain rule.

$$\frac{\partial J}{\partial \mathbf{o}} = \frac{\partial J}{\partial L} * \frac{\partial L}{\partial \mathbf{o}} = \frac{\partial L}{\partial \mathbf{o}} \tag{3}$$

For a regularization function s, it is possible to compute its derivative with respect to the parameters $W^{(1)}$ and $W^{(2)}$.

$$s = \frac{\lambda}{2}\left(\left\|W^{(1)}\right\|_F^2 + \left\|W^{(2)}\right\|_F^2\right) \tag{4}$$

$$\frac{\partial s}{\partial W^{(1)}} = \lambda W^{(1)} \quad \text{e} \quad \frac{\partial s}{\partial W^{(2)}} = \lambda W^{(2)} \tag{5}$$

From that, the gradient of the loss function in terms of $W^{(2)}$ can be obtained.

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial J}{\partial \mathbf{o}} * \frac{\partial \mathbf{o}}{\partial W^{(2)}} + \frac{\partial J}{\partial s} * \frac{\partial s}{\partial W^{(2)}} = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^{\mathrm{T}} + \lambda W^{(2)} \tag{6}$$

In order to compute the gradient of the objective function with respect to $W^{(1)}$, it is necessary to continue the backpropagation in the direction of the input layer of the ANN. With respect to the output of the intermediate layer, **h,** it is as follows**.**

$$\frac{\partial J}{\partial \mathbf{h}} = \frac{\partial J}{\partial \mathbf{o}} * \frac{\partial \mathbf{o}}{\partial \mathbf{h}} = {W^{(2)}}^{T} \frac{\partial J}{\partial \mathbf{o}} \tag{7}$$

In addition, the function ϕ is computed element-wise in the vector of inputs.

$$\frac{\partial J}{\partial \mathbf{z}} = \frac{\partial J}{\partial \mathbf{h}} * \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}) \tag{8}$$

Finally, the gradient with respect to the initial network parameter can be obtained.

$$\frac{\partial J}{\partial W^{(1)}} = \frac{\partial J}{\partial \mathbf{z}} * \frac{\partial \mathbf{z}}{\partial W^{(1)}} + \frac{\partial J}{\partial s} * \frac{\partial s}{\partial W^{(1)}} = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^{\mathrm{T}} + \lambda W^{(2)} \tag{9}$$

Thus, in a generic way, the backpropagation algorithm consists in, during the feed-forward step of the neural network, storing both the values of the activation functions and the derivatives of the activation functions in each node, so later these derivatives can be used in a chain rule to derive the gradient of the objective function with respect to each of the network parameters. With the gradient, then, it is possible to iteratively update the weights. This method allows a complex gradient of a function formed by the combination of several activation functions to be computed in a simple way, by integrating the calculation of the intermediate derivatives in terms of the parameters with the ANN operation (ZHANG et al., 2021).

3.2.2. Principal Component Analysis (PCA)

One of the most traditional and widely used methods for unsupervised learning problems derives deeply from multivariate statistics and is called **Principal Component Analysis** (PCA) (HOTELLING, 1933). It is a multivariate technique that receives a dataset and extracts orthogonal latent variables, called **principal components**. The new set of variables if formed by linear combinations of the original variables, and it is obtained from the covariance matrix of the data, oriented in a way that it maximizes its explained variance. The main objectives of PCA are (1) to extract the most important information from the dataset; (2) to maintain only the most important characteristics of the system under analysis, in order to reduce the dimension of the dataset; (3) to simplify its description and (4) to facilitate the analysis of the structure of the observations and the variables (ABDI; WILLIAMS, 2010).

PCA is one of the classic dimensionality reduction methods, being also widely applied to fault and outlier detection in industrial processes. From the new dataset, composed only by a few principal components, PCA allows easily computing of statistics such as Hotelling's T^2, frequently used as a multivariate statistics based-distance metric from a specific point to the main PCA model. In a complementary way, from the transformation of a specific observation by a PCA model, it is possible to compute the residuals between the original observation and its projection into the hyperplan defined by the principal components, a statistic known as the **sum of squared residuals**, or **Q statistic**. This statistic is frequently applied together with Hotelling's T^2 for fault or outlier detection (ABDI; WILLIAMS, 2010; BRO; K. SMILDE, 2014).

In this work, PCA will be used as a baseline method, with fault detection performed using both T^2 and Q statistics. The reason is that it is commonly used for chemical process monitoring. The results will be compared with the results of the models obtained from reinforcement learning methods.

3.3. REINFORCEMENT LEARNING

The fundamental elements of reinforcement learning (RL) problems can be formalized using the concept of **Markov Decision Processes** (MDP), one of the main formalisms for sequential decision planning, in which the objective is to control a system described by an environment modeled as a set of states and control actions. This objective must be achieved by the maximization of a performance criterion, and MDP is used for several learning problems in stochastic domains (WIERING; VAN OTTERLO, 2012).

An **agent** (the instance responsible for making decisions in the process) is responsible for influencing a probabilistic system by choosing a sequence of **actions**. In each decision epoch, i.e., time steps in which the decisions are made, the system is characterized by its **state** $S_t \in \mathcal{S}$ and, based on the state information, the agent defines, deterministically or stochastically, an action $A_t \in \mathcal{A}(s)$ ($\mathcal{A} = \cup_{s \in \mathcal{S}} \mathcal{A}(s)$) among the set of possible ones (PUTERMAN, 2014).

At certain time step t, the agent receives a representation of the state of the **environment**, $S_t \in \mathcal{S}$ and selects an action $A_t \in \mathcal{A}(s)$. As a consequence of the action performed, the agent receives a numeric **reward,** $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ [1], that implicitly specifies the objective, indicating the direction in which the system should be controlled. In addition, the system transitions to a new state S_{t+1}. In a finite MDP, the sets $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ have a finite number of elements and the variables R_t and S_t have probability distributions that depend only on the action and state at the immediately preceding timestep ($t-1$). When this is true, it is said that the system has the Markovian property. (SUTTON; BARTO, 2018; WIERING; VAN OTTERLO, 2012). Thus, we have equation (10), for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$ e $a \in \mathcal{A}(s)$:

$$p(s', r|s, a) \equiv \Pr\{S_t = s', R_t = r \mid S_{t-1} = s,\ A_{t-1} = a\} \tag{10}$$

The probabilistic function p, therefore, defines the dynamic of the process. **Figure 6** illustrates the interaction between agent and environment in an MDP.

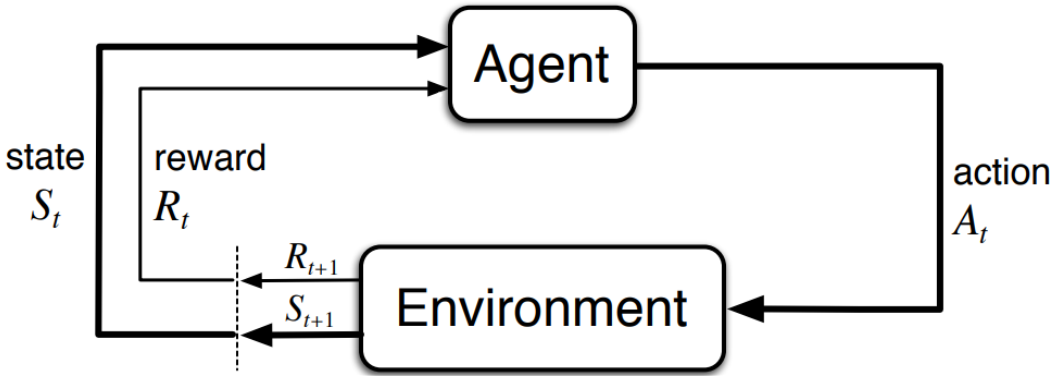


Figure 6: The agent-environment interaction in a Markov Decision Process (MDP) (SUTTON; BARTO, 2018)

[1] Some authors define the immediate reward for taking action A_t while in state S_t as R_t, i.e., the reward belongs to the same time step as the action taken. Both definitions are found in the literature.

The **policy** is defined as the behavior of the agent in certain timestep, and it is a mapping from the possible states of an environment to the actions that need to be selected when said environment finds itself in each of these states. The functions that represent a policy can be simple or complex, deterministic or stochastic, and can demand great computation capacity (SUTTON; BARTO, 2018). In a MDP, the policy π will return, for each state $s \in \mathcal{S}$, an action $a \in \mathcal{A}$, and can be defined deterministically, $\pi : S \rightarrow A$, or stochastically, $\pi : S \times A \rightarrow [0,1]$. (WIERING; VAN OTTERLO, 2012).

To search for optimal policies, it is necessary to understand what defines this concept. For problems such as MDP and reinforcement learning (RL), the objective of the agent, following a policy, is to collect rewards. In case the only consideration is the immediate reward, $\mathbb{E}[r_t]$ is a simple optimization objective. Usually, however, there is also the concern with future rewards, that can be considered in three ways: (1) a finite horizon of rewards $\left(\mathbb{E}[\sum_{t=0}^{h} r_t]\right)$; (2) an infinite horizon of discount rewards ($\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$), $\gamma \in [0,1]$ being a discount factor used to weight the future rewards; or (3) the average reward ($\lim_{h\to\infty} \mathbb{E}\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right]$) (WIERING; VAN OTTERLO, 2012).

Value functions are a common way of connecting the optimization criteria to the function that defines the agent's policy. These functions provide estimates of how advantageous it is, in terms of expected rewards, to be in certain state or to select certain action in a given state (WIERING; VAN OTTERLO, 2012). The value function, therefore, estimates the value of a state, i.e., the total reward the agent should accumulate in case it follows its current policy having this state as the starting point. Unlike the reward, which indicates how desirable an action will be in a given state, the value represents the long-time desirability of the state. For a state in which, independently of the action taken, the outcome is always a low reward, we know for sure that this state will provide a low immediate reward. However, it is possible that this same state has a high value, as long as it is followed by states that, according to the current policy, will result in large rewards (SUTTON; BARTO, 2018).

Sutton and Barto (2018) also suggest the definition of G_t as the expected future rewards, for both finite and infinite horizons, regardless of the rewards being discounted or not. For a hypothetical case in which we have an infinite horizon of discounted rewards, the value function can be defined according to Equation (15), for all $s \in \mathcal{S}$.

$$v_\pi(s) \equiv \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \qquad (11)$$

The function v_π is called the **state-value function**, and $\mathbb{E}_\pi$ is the expected value of any variable while under a policy π.

The **action-value function** is defined as the value that an agent has in performing a given action a in a given state s, while following a policy π (SUTTON; BARTO, 2018). This function is defined in a similar way as the state-value function, according to Equation (12), and the action-value is also known as **q-value**.

$$q_\pi(s,a) \equiv \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \quad (12)$$

Many methods, e.g., dynamic programming, rely on the existence of a **model** for policy optimization in an MDP. A model allows inferences to be made regarding the system's behavior, e.g., the prediction of the next state-reward pair, and it is used for planning the decisions in the process. In a reinforcement learning context, methods that use models for problem solving are **model-based methods**, as opposed to **model-free methods**, which use trial-and-error techniques, collecting experience and evaluating the results over a series of trajectories. Some methods which combine both are also present in literature (SUTTON; BARTO, 2018).

One of the most important dilemmas in RL is the **exploration-exploitation trade-off**, which indicates the need to balance exploration and exploitation during the development of a learning algorithm. **Exploration** is the need for the agent to experiment several actions in different contexts, aiming to acquire more knowledge about the system and its dynamics and, perhaps, find even better actions than the ones already presented. **Exploitation**, on the other side, refers to the act of choosing the best possible actions (indicated, for example, by a high action-value) in order to maximize the future rewards (WIERING; VAN OTTERLO, 2012).

Wiering and van Otterlo (2012) also talk about the ε-greedy policy exploration strategy, one of the most relevant techniques used to ensure RL agents will explore the environment. In this strategy, the agent selects the best possible action with probability $(1-\epsilon)$ and another random action with probability ϵ. Other exploration methods, such as Boltzmann's, similar to ϵ-greedy policy, but with probabilities weighted by its respective q-values (from the action-value function) are widely used.

3.3.1. Deep Q-Learning

3.3.1.1. Q-Learning

The Q-Learning algorithm (WATKINS, 1989) is a reinforcement learning method from the model-free class of algorithms, i.e., a method in which an agent estimates the state-value and action-value functions solely from experience, gathered along multiple interactions with an environment, without knowing the dynamics of the model that describes this environment.

Watkins and Dayan (1992) explain that, in this method, an agent receives data representing a state s ($\in \mathcal{S}$) and picks a specific action a ($\in \mathcal{A}$) according to the current policy. After the action is taken, the agent receives a probabilistic reward r($\in \mathcal{R}$), whose mean value $\bar{r}_s(a)$, assuming the Markovian property, depends only on the current state and action. The following state, s' ($\in \mathcal{S}$), is given in a similar way as the transition probability from Equation (10), according to Equation (13):

$$p(s'|s,a) \equiv \Pr\{\mathrm{S}_{t+1} = \mathrm{s}'|\mathrm{S}_t = \mathrm{s}, \mathrm{A}_t = a\} \tag{13}$$

The agent's objective is to find an optimal policy, i.e., a policy that maximizes the total discounted reward, which means the rewards from future time steps are not as important as the current ones. In given time step t, the following rewards will be discounted by a factor γ^n, $(0 < \gamma < 1)$, with S_{t+n} being the state at n time steps ahead of t (WATKINS; DAYAN, 1992). From this definition, following a policy π, Equation (14) defines the value of a state s as:

$$v_\pi(s) \equiv \bar{r}_s\big(\pi(s)\big) + \gamma \sum_{s'} p(s'|s,\pi(s)) v_\pi(s') \tag{14}$$

Watkins and Dayan (1992) further explain that the recursive equation takes into account the immediate reward as a result of the action determined by policy π. It then transitions, with probability $p(s'|s,\pi(s))$, to a new state with value $v_\pi(s')$, which will be weighted by the discount factor γ. Additionally, the authors define the q-value, that is, the expected discounted reward after executing action a in state s in Equation (15), in a similar way as it was performed before in Equation (12):

$$q_\pi(s,a) \equiv \bar{r}_s(a) + \gamma \sum_{s'} p(s'|s,\pi(s)) v_\pi(s') \quad (15)$$

The objective of Q-Learning is precisely the estimation of q-values (or action-values) for an optimal policy, defined as $q^*(s,a) \equiv q_\pi^*(s,a), \forall\, s,a$. It is important to note that the theory of dynamic programming ensures the existence of at least one stationary policy π^* that guarantees, as a result of the optimal value function, the maximum expected cumulative reward from any state, shown in Equation (16) (WATKINS; DAYAN, 1992).

$$v^*(s) \equiv v_\pi^*(s) = \max_a \left\{ \bar{r}_s(a) + \sum_{s'} p(s'|s,\pi(s)) v_\pi^*(s') \right\} = \max_a \{q^*(s,a)\} \quad (16)$$

Thus, in case the agent is able to learn the q-values from the optimal policy, it is possible to select the actions that yield the best return. The learned action-values are presented as Q.

The idea behind the method relates to Monte Carlo (MC) methods, which provide a foundation for estimating value functions from a collection of experiences. Through a sequence of finite episodes, in which the agent visits several states and selects different actions, it is possible to collect, with enough experience, reward values for all state-action pairs in the process. With this, value functions can be estimated by averaging the accumulated discounted rewards from each episode, which tends to converge to the expected value after sufficient attempts (SUTTON; BARTO, 2018).

Another fundamental idea that provides a foundation for the Q-Learning algorithm is Temporal Difference Learning, which, as the Monte Carlo methods, can be used to estimate value functions from experience. For MC methods, V, which is an estimate for the state-value function v_π, can only be incremented by the end of each episode (which is composed by several time steps), with G_t, the cumulative reward actually received (and not estimated) from time step t to the end of the episode, according to Equation (17) (SUTTON; BARTO, 2018).

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \quad (17)$$

Therefore, according to this equation, the update of the estimate ($V(S_t)$) is performed based on the difference of this previous estimate and the real value obtained (G_t), proportionally to a constant parameter α, the learning rate. For Temporal Difference methods, however, the value $V(S_t)$ can be incremented at each time step, according to Equation (18).

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} - \gamma V(S_{t+1}) - V(S_t)] \quad (18)$$

The value update is performed immediately after the transition to state S_{t+1} and the output of the reward R_{t+1}. In this case, the update is made with reference to $R_{t+1} - \gamma V(S_{t+1})$ and its difference to the value estimated previously. This method is known as $TD(0)$ (SUTTON; BARTO, 2018).

Wiering and van Otterlo (2012) define that, for Q-Learning, the objective is to obtain an estimate Q for the action-value functions, q_π, and this estimate is updated according to an adaptation of TD Learning, as shown in Equation (19).

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (19)$$

Sutton and Barto (2018), furthermore, guarantee that the action-values (Q) converge to those of the optimal action-value function (q^*) independently of the policy π, which allowed convergence proofs to be established. For convergence to happen, the only requirement is that the values of all state-action pairs are continuously updated, even though the current policy will determine which pairs are being visited at each time step. Thus, the method builds a Q-value table which contains all state-action pairs, and then it is able to consult the best possible action at each time step.

3.3.1.2. Deep Q-Learning

Several reinforcement learning methods were successful in solving some types of classical problems modeled by MDPs. Even though convergence proofs were established for Q-Learning, this algorithm faces some issues, mainly: (1) for a large number of states, the look-up table built to store the Q-values of all state-action pairs can face the issue famously known as **curse of dimensionality**; (2) the existence of possible correlations between the sequence of observations; (3) the fact that small updates in Q can lead to significant changes in the policy and,

therefore, in the data distribution; and (4) the existence of correlations between the action-values Q and the target update values ($r + \gamma \max_{a'} Q(s', a')$) (MNIH et al., 2015; WU et al., 2022).

To reduce the impact of these issues in terms of the convergence and stability of RL algorithms, a variant of Q-Learning, named **Deep Q-Learning** (DQL), was proposed by Minh et al. (2015). This variant uses deep neural networks called Deep Q-Networks as substitutes for the Q-values table and is based in two main solutions: (1) the introduction of an **experience replay buffer**, responsible for storing the past agent-environment interactions, as well as collecting a random sample of past observations and using it for updating the network parameters, which eliminates the correlations from the sequence of observations and makes the changes in the data distributions smoother; (2) the introduction of target values that are separately calculated and whose updates occur only periodically, instead of simultaneous updates in each iteration, which significantly decreases the correlation between the output and target values.

In the algorithm, the agent's experiences, $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$, are stored, at each time step, in a dataset $D_t = \{e_1, \dots, e_t\}$ and, during the learning process, random samples of experiences are uniformly collected, $(S_t, A_t, R_{t+1}, S_{t+1}) \sim \mathrm{U}(D)$, and used for the updates, instead of the sequence of timesteps belonging to the previous episode. To independently obtain the target values, a second network, called Target Q-Network, is introduced. The loss function for parameter updates during a given iteration i is represented in Equation (20),

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right], \qquad (20)$$

in which γ is the discount factor, θ_i are the Q-Network parameters, and θ_i^- are the target network parameters, updated every C decision epochs (MNIH et al., 2015).

Therefore, instead of the traditional method used to estimate the Q-values in Q-Learning, in which for each different sequence of states a single Q-value will be separately estimated, the action-value function will be estimated by a neural network parameterized by θ, in a such a way that $Q(s, a; \theta) \approx Q^*(s, a)$. Furthermore, the optimal target values, previously obtained by $r + \gamma \max_{a'} Q^*(s', a')$, will be estimated with $r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ by the target network (MNIH et al., 2015).

Usually, the loss function optimization is performed with Stochastic Gradient Descent, and the gradient of the loss function is obtained from the differentiation of the loss function (Equation (20)) in terms of the weights, i.e., the set of parameters θ, as shown in Equation (21).

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q(s,a;\theta_i^-) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right] \qquad (21)$$

it is also important to note that the target network parameters θ_i^- are independent from θ, which are being optimized (MNIH et al., 2015).

3.4. USUAL REINFORCEMENT LEARNING APPLICATIONS

Reinforcement learning emerged as a way of building algorithms with the capability of learning optimal policies for decision making and performing specific tasks based on a sequence of states, actions and rewards. In the fields of robotics and control, several methods are employed to solve complex problems. Martinez-Marin (2006) proposed a motion planning algorithm for non-holonomic robots. Mordatch et al. (2016) developed an automatic method for interactive controlling humanoid robots, combining model-based policy search and online model learning methods. Bagnell and Schneider (2001) developed a model, also based on policy search, for flying an autonomous helicopter.

Traditionally, several advancements and contributions for the state-of-the-art have been obtained in the development of reinforcement learning methods for playing games. Tesauro (1994) developed TD-Gammon, an algorithm that achieved excelent performance in gammon, built based on temporal difference learning. In 1997, the Deep Blue supercomputer defeated the grandmaster Garry Kasparov in a chess match (HSU, 1999). Thiery and Scherrer (2009) developed an algorithm capable of playing the famous game of Tetris, creating an agent capable of cleaning 35 million lines per game, a result far beyond human capabilities. Silver et al. (2017) developed AlphaGo-Zero, a computer program based solely on reinforcement learning, without the need for pre-existing human knowledge implementations, and obtained a 100% victory rate in Go, a traditional Chinese game.

In addition, there are several reinforcement learning applications in fields related to medicine, biology and health studies. Some of these applications consist in agents capable of adapting medication dosages for patients in cancer treatment, such as the one developed by Tseng et al. (2017), in which the authors

proposed a method based on a Deep Q-Network (DQN) for adaptive treatment in radiotherapy, and Zhao, et al. (2009), in which a recommendation system for medication dosages for chemotherapy treatments was developed. For patients with diabetes, Noori, et al. (2017) used SARSA, a temporal difference method, to control insulin levels.

Similarly to fault detection and diagnosis in industrial processes, medical applications of RL that aim to detect and diagnose medical conditions are widely developed. Ghesu et al. (2016) presented a method based in Deep Reinforcement Learning (DRL) to automatically detect anatomical landmarks using medical images, obtaining superior performance if compared to several traditional supervised learning methods. Maicas et al. (2017) used a DRL in order to automatically detect breast lesions from magnetic resonance imaging (MRI).

3.5. APPLICATIONS OF REINFORCEMENT LEARNING FOR FAULT DETECTION AND DIAGNOSIS IN CONTINUOUS CHEMICAL PROCESSES

The most traditional applications of RL in the chemical process industry refer to process control. Lee and Lee (2005) implemented J-Learning and Q-Learning for the optimal control of a Continuous Stirred-Tank Reactor (CSTR). Furthermore, Joy and Kaisare (2011) developed a Q-Learning based method for optimal control of a Plug Flow Reactor (PFR). Jiang et al. (2018) implemented the optimal control of an industrial flotation plant using the Interleaved Learning method, and it was capable of online learning without the need of knowing the dynamics of the flotation process.

Some RL applications for fault detection and diagnosis in industrial setting have been published recently, with the majority of them focusing on rotating machinery. Wu et al (2022) developed a neural network named Deep Reinforcement Transfer Convolutional Network (DRTCNN) to diagnose faults in industrial rolling bearings for a dataset with limited labels. They achieved superior results compared to all peer-reviewed benchmark methods presented by transferring parameters from a pre-trained RL model from a similar dataset.

Given the safety, financial and quality requirements, real industrial plants tend to spend much more time operating within normal conditions than under fault regimes. Zhong, et al. (2023), with the objective of detecting and diagnosing faults in rotating machinery from the nuclear industry, explored the capability of RL algorithms to deal with datasets with such class imbalance characteristics. By designing a reward function inversely proportional to the number of samples of the specific fault being diagnosed, the authors developed a model more robust to class imbalance, compared to other models chosen for comparison purposes.

Qian and Liu (2022) developed two types of DQN (the first based in Convolution Neural Networks, CNNs, and the second in Gated Recurrent Units, GRUs) for fault diagnosis in rotating machinery. The authors developed models whose performance was superior to the baseline models (Support Vector Machines – SVMs – and supervised learning applications of CNNs and GRUs), with an accuracy of over 99%. Wang and Xuan (2021) implemented an improved actor-critic algorithm also for diagnosing faults in rotating machinery, with superior performance to classical Deep Learning methods.

Ding et al. (2019) implemented a DQN without random parameter initialization. For that, the authors first trained a Stacked Autoencoder (SAE), that aims to represent an input in a latent space and then to reconstruct the inputs from these representations, and used the parameters of the first layers of the SAE for initializing a CNN that computed the action-value function in a Deep Q-Learning framework. The method was applied to fault diagnosis in bearings and hydraulic pumps and obtained similar results as some traditional supervised learning approaches. For a more complex dataset, composed of 10 different types of faults gathered in 3 different frequencies, the method had superior performance.

A common characteristic of fault detection and diagnosis methodologies for rotating machinery is the use of low-dimensional data, often univariate. For multivariate systems, in an approach that used a fuzzy system as a function approximator as opposed to artificial neural networks, Malik et al. (2020) presented a fault diagnosis framework for power transformers based on dissolved gas analysis (DAG) data, gathered from oil found in real power transformers. The authors used the J48 algorithm to select the 8 most appropriate variables (among 24 total) and then developed a Fuzzy Q-Learning method which obtained superior results when compared to the baselines (SVM, ANN and fuzzy logic).

To the best of the author’s knowledge, there are no published RL applications for fault detection and diagnosis using multivariate data from continuous chemical processes, which indicates significant value in developing a tailored framework for such purposes.

4. CASE STUDY (TENNESSEE EASTMAN PROCESS)

The Tennessee Eastman Process (TEP) (DOWNS; VOGEL, 1993) is a realistic simulation based on a real industrial process. Even though there were alterations in the components, kinetics and operating conditions of the process, due to proprietary reasons, the fact that it is not a contrived problem stands out and this simulation, developed by the process control group of Eastman Chemical Company alongside researchers from the University of Tennessee, still is one of the most important academic benchmarks for studying process fault detection and diagnosis systems, control technologies, being applicable to a wide variety of process issues.

The process has two products and one byproduct, obtained from four gaseous reactants and also an inert, making a total of eight components (A, B, C, D, E, F, G and H). The reactions featured in the process are irreversible, exothermic and approximately first-order with respect to the concentrations. They are listed below.

$$A(g) + C(g) + D(g) \rightarrow G(liq), \qquad Product\ 1$$

$$A(g) + C(g) + E(g) \rightarrow H(liq), \qquad Product\ 2$$

$$A(g) + E(g) \rightarrow F(liq), \qquad Byproduct$$

$$3D(g) \rightarrow 2F(liq), \qquad Byproduct$$

All reaction rates relate to temperature through an Arrhenius expression, and the reaction to produce G has the highest sensitivity to temperature.

In the original problem setting, given that it was first designed as a benchmark for process control methodologies, Downs and Vogel (1993) built the simulation as an open-loop process. For FDD purposes, as real chemical processes operate under closed-loop control, a plant-wide control scheme is needed for accurately developing a framework. There are many works regarding plant-wide control structures for the TEP, and the one implemented by Lyman and Georgakis (1995) was chosen, for being the one commonly used. Furthermore, Xavier and De Seixas (2018) modified the code for this closed-loop setting to allow the simulation of a more realistic industrial operation, regarding class imbalance. For implementation purposes, this project relied on Tep2py, a Python interface to wrap the original Fortran program of TEP, provided by Xavier and De Seixas (2018).

Figure 7 shows a diagram of the closed-loop operation of TEP. The process is composed of five major unit operations: a reactor, a product condenser, a vapor-liquid separator, a recycle compressor and a product stripper. The reactants are fed to the reactor and then form liquid products. Dissolved in the liquid phase, there is a nonvolatile catalyst for the gas phase reactions, which are all exothermal and, thus, there is also an internal cooling bundle for removing the heat of reaction. Both the products and the unreacted feeds leave the reactor as vapors, and this stream goes through a cooler in order to condense the product. After the condenser, the stream passes through a vapor-liquid separator, which outputs two new streams, the first of which is composed by the non-condensed products. This stream is recycled through a centrifugal compressor to enter the reactor again. The second stream leaving the condenser is composed of the condensed components, which move to a product stripping column, and stream 4 is used to strip and remove the remaining reactants. The desired products G and H are removed from the stripper base and their separation happens in a stage which is not included in this problem. The inert and byproduct are purged from the system after the vapor-liquid separator.

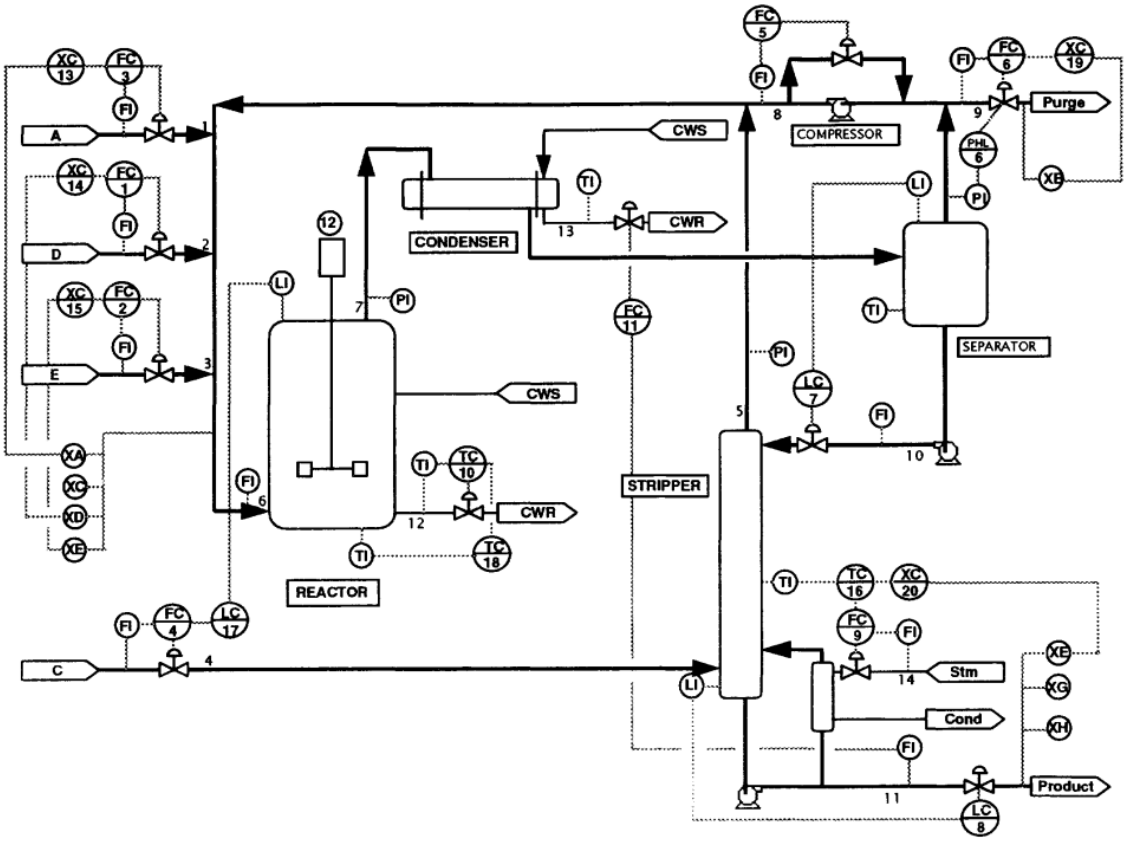


Figure 7: Tennessee Eastman Process diagram (LYMAN; GEORGAKIS, 1995).

The simulation outputs process data composed of a total of 53 variables (41 measured variables and 12 manipulated variables). However, for the current project, only 52 will be used, since one of them, which refers to the agitator speed, remains constant. Also, it is possible to introduce 20 different process disturbances, by coded variables called IDVs, representing a set of tests that can be used for comparison purposes. The disturbances come from five different categories: step, random variation, slow drift, sticking and unknown. These are the faults to be detected. **Table 1** lists all process disturbances, and they can be inserted at any time, for any duration desired.

Table 1 – Process disturbances (faults) from the Tennessee Eastman Process simulation (DOWNS; VOGEL, 1993).

Variable	Process variable	Type
IDV (1)	A/C feed ratio, B composition constant (stream 4)	Step
IDV (2)	B composition, A/C ratio constant (stream 4)	Step
IDV (3)	D feed temperature (stream 2)	Step
IDV (4)	Reactor cooling water inlet temperature	Step
IDV (5)	A feed loss (stream 1)	Step
IDV (6)	C header pressure loss – reduced availability (stream 4)	Step
IDV (7)	A, B, C feed composition (stream 4)	Random variation
IDV (8)	D feed temperature (stream 2)	Random variation
IDV (9)	D feed temperature (stream 2)	Random variation
IDV (10)	C feed temperature (stream 2)	Random variation
IDV (11)	Reactor cooling water inlet temperature	Random variation
IDV (12)	Condenser cooling water inlet temperature	Random variation
IDV (13)	Reaction kinetics	Slow drift
IDV (14)	Reactor cooling water valve	Sticking
IDV (15)	Condenser cooling water valve	Sticking
IDV (16)	Unknown	Unknown
IDV (17)	Unknown	Unknown
IDV (18)	Unknown	Unknown
IDV (19)	Unknown	Unknown
IDV (20)	Unknown	Unknown

To represent the impact of the faults in the process data, nine arbitrary variables (seven being measured and two being manipulated closed-loop ones) were selected in both normal and faulty operation for the four different faults, and the respective plots are presented from **Figure 8** to **Figure 11**. These selected

variables are only for illustration purposes, and the fault detection framework will contain all fifty-two measured and manipulated variables.

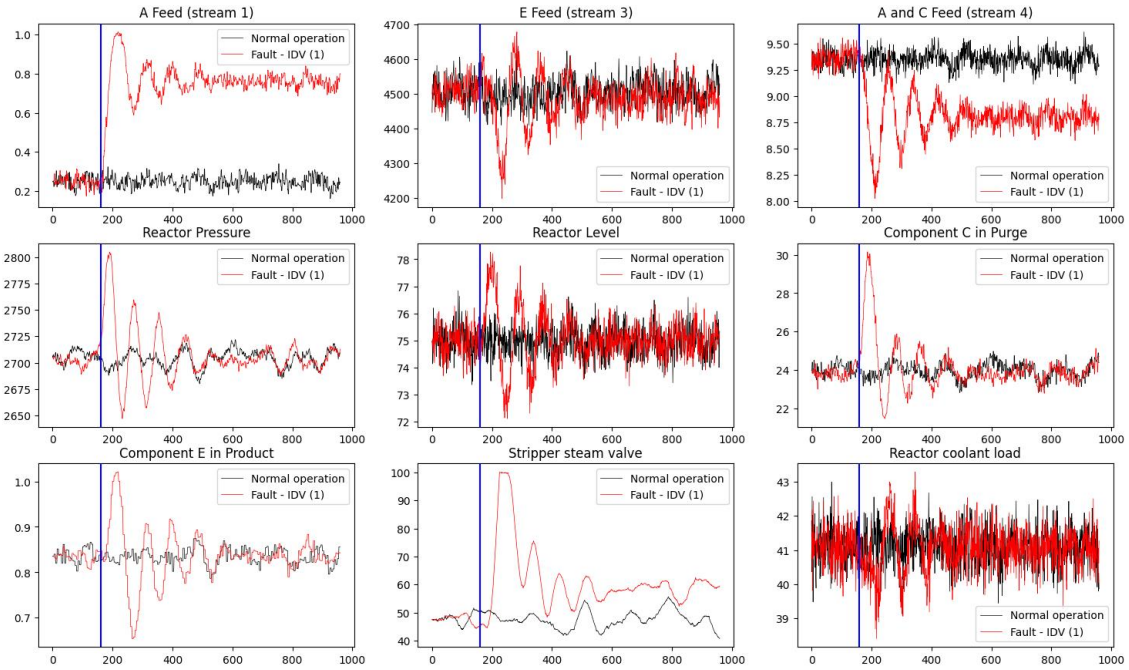


Figure 8: TEP plots for nine variables under both normal and fault 1 operations. Data was gathered from 48h (960 observations) of simulated process, with the fault starting after the first 8h (160 observations), as indicated by the blue vertical line in each plot.

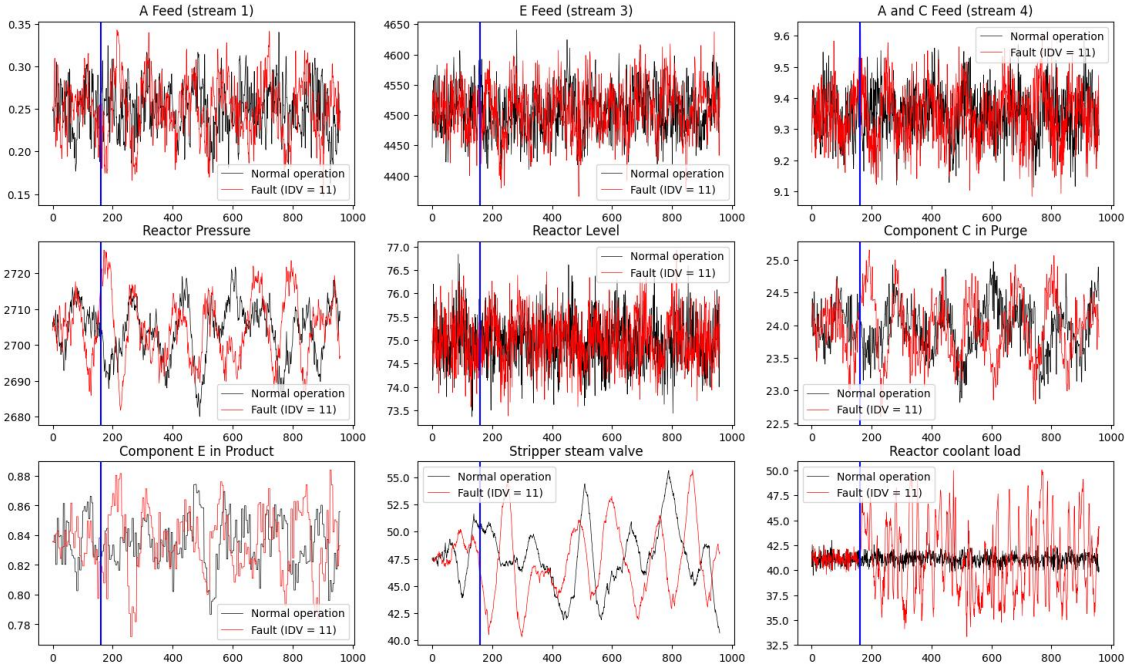


Figure 9: TEP plots for nine variables under both normal and fault 11 operations. Data was gathered from 48h (960 observations) of simulated process, with the fault starting after the first 8h (160 observations), as indicated by the blue vertical line in each plot.

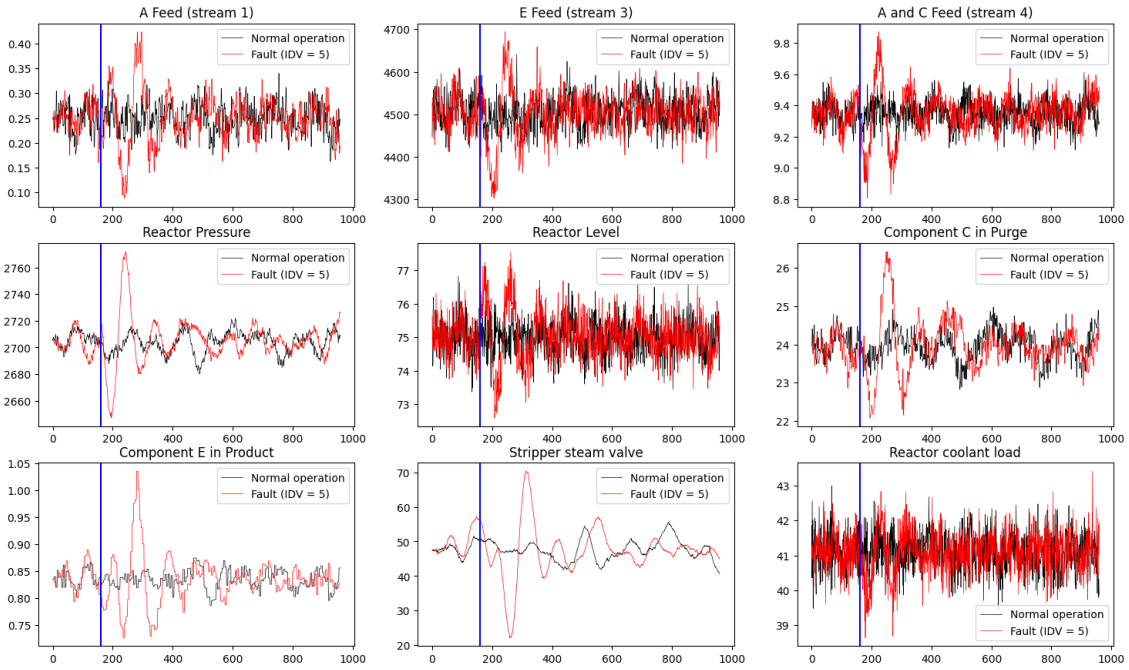


Figure 10: TEP plots for nine variables under both normal and fault 5 operations. Data was gathered from 48h (960 observations) of simulated process, with the fault starting after the first 8h (160 observations), as indicated by the blue vertical line in each plot.

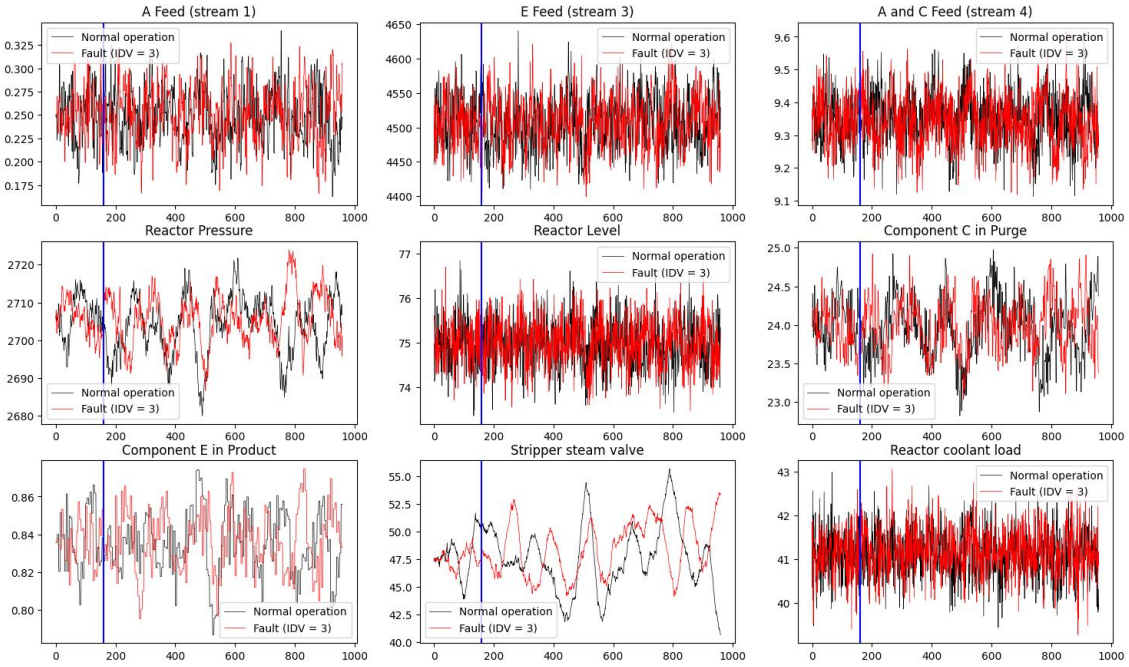


Figure 11: TEP plots for nine variables under both normal and fault 3 operations. Data was gathered from 48h (960 observations) of simulated process, with the fault starting after the first 8h (160 observations), as indicated by the blue vertical line in each plot.

By qualitatively analyzing the figures above, we can notice that some faults form clearer patterns than others, and this might impact the difficulty of the detection task. For instance, taking fault 1 as an example, we know from **Table 1** that it represents a step-type fault in the feed stream ratio between components

A and C. By looking at the first plot in **Figure 8**, we can easily see the step occurring in the A Feed variable after the fault starts. For other faults, however, the patterns may not be as clear.

5. METHODOLOGY

In this section, the methodology used to develop and test a reinforcement learning framework for fault detection in continuous chemical processes will be described. First, the adaptation of the fault detection task from the usual supervised learning to the proposed RL approach is discussed. Then, the design of a custom environment and its particular interaction with the agent is presented, including some information regarding programming languages and class structure. Then, the RL method used for developing the agent is explained and important information, such as hyperparameter settings and training structure, is presented. Finally, the performance metrics chosen for evaluating the agent, as well as the validation and testing steps of the framework are listed, together with the complete framework developed for this dissertation.

Fault detection is usually a supervised learning task under the branch known as classification, in which the learning process focuses in determining the mapping between input process data and discrete labels called classes, in this case the fault types. Therefore, in order to develop a reinforcement learning framework that creates an agent capable of performing fault detection in continuous chemical processes, it is necessary to adapt the task into a more suitable approach. This approach can be defined as a type of fault detection “game”, in which the agent receives, at each timestep, the state of the environment, i.e., a matrix in which the columns are the process variables and the rows are the observations, and picks an action, which means to select the corresponding label, i.e., to detect or not the fault. **Figure 12** shows a general flowchart of a fault detection game.

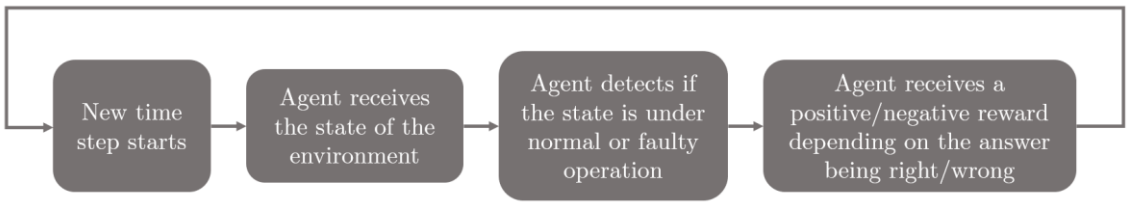


Figure 12: General flowchart of a fault detection game.

The goal of the present work is to design and test a framework capable of training an agent to achieve good results in fault detection in continuous processes. The Deep Q-Learning algorithm was selected as the core of the framework, mainly for its formally proven ability to perform well in discrete action-spaces (WATKINS; DAYAN, 1992), which is the case for fault detection.

For this methodology, the Python programming language was the main development tool, and, thus, firstly it was needed to wrap the original Fortran 77 code of the TEP using Tep2py, a Python interface to the program with the modifications implemented by Xavier and De Seixas (2018), previously discussed in Section 4. After the code was wrapped, a custom RL environment was developed with class inheritance to the base environment class provided by the Gymnasium (formerly developed as OpenAI Gym) open-source library. This custom environment (CE) consists of four different aspects: (1) the **TEP simulation**, which outputs process data at each timestep; (2) a **sliding window**, which will be the actual input of the DQN; (3) the **step function**, which receives the action selected by the DQN and evaluates the action in terms of the numeric reward; (4) the **reset function**, which resets all of the environment to its default settings. The DQN from the Deep Q-Learning method represents the agent, and thus the CE needs to provide the structure necessary for the agent-environment interaction to happen.

The **TEP simulation** is the Python-wrapped Fortran 77 code, and the data is generated according to a $N \times 20$ matrix, in which the columns represent the possible IDVs (or faults), and the rows are the number of observations desired. For normal operation simulations, this matrix should be composed only of zeros. In case a fault needs to be inserted for a certain range of timesteps, the matrix elements from the column that represents the desired fault, during the range of observations in which the fault will happen, should be set to 1. The simulation outputs process data in a $N \times 52$ shape, for N observations and 52 variables, with a 3-minute interval (Russel et al. (2000)) between two sequential observations. The **sliding window** is a fixed-length $w \times 52$ window of observations that is the input to the DQN as the state of the environment for a specific timestep. The window slides through the data with a stride of 1 observation, meaning that the window in the first timestep is composed of observations $1{:}\,(w)$, the second is composed by observations $2{:}\,(w+1)$, the third by observations $3{:}\,(w+2)$ and so on, in a way that, at each new timestep, the oldest observation is dropped in order to include the newest one.

The **step function** is the method (as defined in object-oriented programming) responsible for receiving the action picked by the agent and evaluating it according to the desired output, i.e., whether or not there is a fault and, given that, what was the agent's prediction. After the evaluation, this function also outputs the reward, which was set to –2 for wrong predictions and +1 for right predictions. The decision to set the reward for wrong answers with a greater absolute value than that of the reward for right predictions was intuitively based on the need to minimize the **False Alarm Rate** (FAR) and **Missed Detection Rate** (MDR) (both metrics will be discussed with more detail further in this work) and the need for the agent to be implicitly inclined to avoid getting wrong answers. However, this reward function was designed mainly on a trial-and-error approach,

which indicated faster conversion and better performance when wrong predictions are penalized more severely than right predictions are rewarded. The **reset function**, as said before, is used to reset the simulation parameters, in order to get a default initial environment every time a new episode starts. **Figure 13** shows a schema of the designed CE and its agent-environment interaction.

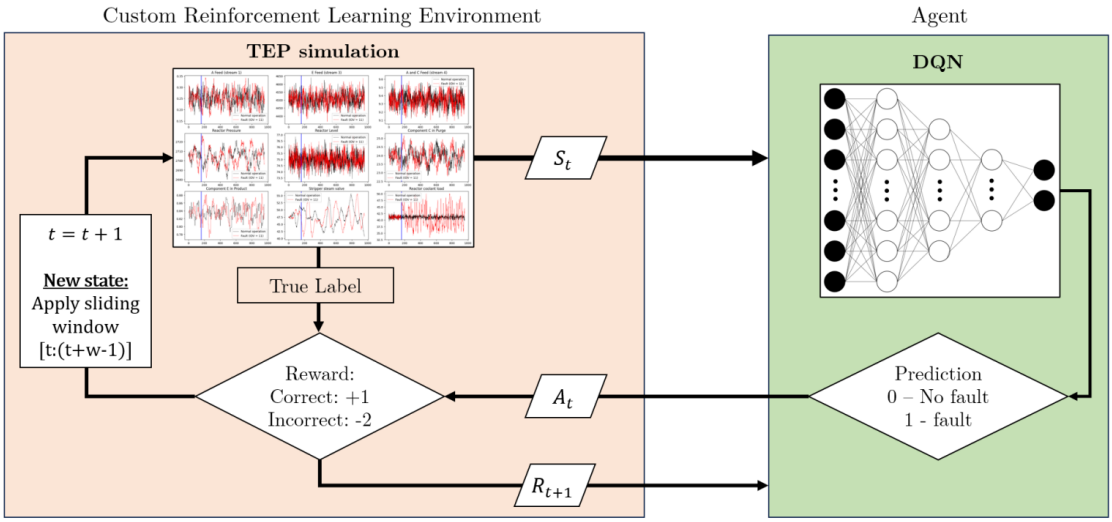


Figure 13: Agent-environment interaction for the custom RL environment designed to perform fault detection in the Tennessee Eastman Process (TEP).

For this work, the training simulations were fixed in a total of 480 observations, which represent a 24h simulation of the industrial plant. Also, the faults start in random time steps with equal probability, and continue to occur until the end of the simulation, with a limit of one fault per episode. Therefore, the episodes have a total number of 480 timesteps, and the sliding window size w was set to 5 observations. These parameters, similarly to the reward function, were set in an intuitive approach after some trials. Whereas smaller windows would provide less information about the transition between normal and fault operations, bigger windows have a negative influence in training time and sampling efficiency and could also have a negative impact on the performance. Therefore, the window size choice was able to balance this trade-off and produce efficient and good results in terms of performance.

The actual training procedure of the agent involves the DQL algorithm, and for this framework, the DQN architecture chosen was a Multi-Layer Perceptron (MLP), the same type of structure shown in **Figure 4(b)**, in section 3.2.1. Two different MLP architectures were evaluated in this present work, both with three hidden layers. Both structures have 128 neurons in the second hidden layer and 64 neurons in the third hidden layer, and they differ only in the dimension of the first hidden layer, the first one having 256 and the second one 64 neurons. The

input of the neural network will be a 5×52 sliding window, which is flattened before being feedforwarded to the ANN, and therefore the input layerhas 260 neurons. Additionally, the output layer has 2 neurons, since fault detection algorithms have two possible predictions (normal operation and fault). For both architectures, the activation function is the ReLU function. The DQL algorithm was implemented using the Stable-Baselines 3 open-source Python library. The hyperparameters of the algorithm are in **Table 2**.

Table 2 - Deep Q-Learning algorithm hyperparameters

Hyperparameter	Definition	Value
Learning rate	Parameter updates will be proportional to this value	0.0001
Buffer size	Maximum amount timesteps that can be stored in the experience replay buffer	48000
Learning start timestep	The timestep in which the algorithm will start updating the DQN parameters. Before that, it will only store experiences	48000
Batch size	Size of the batch used for updating the parameters	480
Train frequency	Train frequency in terms of steps	1
Exploration fraction	The fraction of the learning process in which the value of ϵ will decay from its initial to its final value	0.25
Initial ϵ value	Initial random action probability	1.0
Final ϵ value	Final random action probability	0.05
Target network update frequency	Frequency (in terms of timesteps) in which the Target Q-Network will be updated	10000

With the information from **Table 2**, the training algorithm for fault detection can be defined, and it was performed individually for each of the twenty faults previously described. Therefore, the goal was to train forty different agents, with two different network architectures, to detect twenty different faults. Each episode contains a 24h simulation and its data is available in a 480×52 matrix. The simulation starts as a normal-state operation, and then, at a randomly selected timestep, one fault is inserted and the simulation continues as a fault-state operation. At each timestep, the state of the environment contains only 5 observations, according to the sliding window, and if one of the observations contains a fault, the true label of the operation is considered to be a fault. The state is then feedforwarded by the DQN, which outputs the detection probabilities of both the normal and fault operation. With probability ϵ, according to the ϵ-

greedy policy, the agent selects a random action, otherwise it selects the action with the highest probability. It is crucial to notice that the ϵ parameter starts at 1.0 and decays steadily until 25% of the total episodes are completed, when it reaches 0.05, its minimum. This ensures exploration rates are very high in the beginning of the training procedure, making the agent explore and store these different experiences in the experience replay buffer.

For the first 48000 steps, i.e., 100 episodes, the parameters aren't updated, and the agent only collects experiences and stores it in the buffer, which has the maximum capacity of 48000 experiences. Starting from the 101th episode, the agent starts to learn, i.e., update the DQN parameters, according to the gradient descent method, using a smooth L1 loss function. For each update, the algorithm samples a random batch of 480 experiences from the replay buffer and these experiences are used to update the parameters, weighting each gradient step with the learning rate. For every 1000 timesteps, the parameters of the Target Q-Network are matched with those of the regular DQN. **Figure 14** shows the flowchart of the training process for the proposed framework.

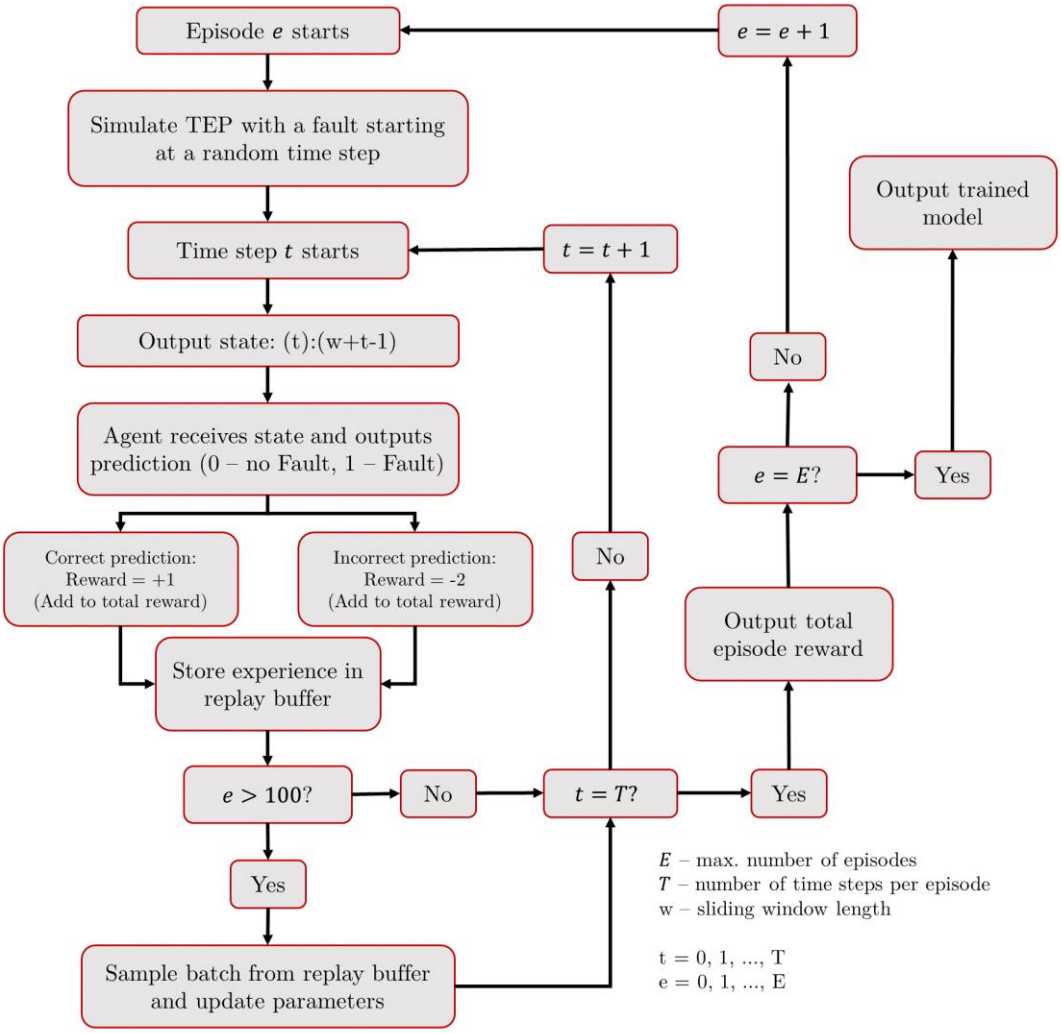


Figure 14: Flowchart of reinforcement learning framework for training a fault detection agent.

To evaluate the trained models, three different performance metrics were selected. Two of those metrics can be defined with a **confusion matrix**, which is a $n \times n$ matrix that, for n classes, contains metrics that are useful to evaluate and analyze classification models. For binary classification problems, the confusion matrix has four possible answers: (1) **true positives (TP)**, which are the positive examples correctly labeled (in this case, the fault samples that were detected as faults); (2) **false positives (FP)**, which are the amount of negative examples labeled incorrectly as positive, i.e., the normal operation samples that were detected as faults (that is, false alarm); (3) **true negatives (TN)** are the amount of negative examples that were correctly classified, i.e., the normal operation samples in which the fault was not detected; (4) **false negatives (FN)**, i.e., the amount of positive examples that were incorrectly classified as negatives, in this case the fault samples for which the fault was not detected (that is, missed detection).

Based on these definitions, the first metric to be evaluated for the trained models is the **False Alarm Rate (FAR).** A false alarm consists of a false positive example, and the rate is defined as the percentage of the normal operation samples that were detected as faults by the model, as shown in Equation (22).

$$FAR = \frac{FP}{TN + FP} = \frac{Normal\ operation\ samples\ labeled\ as\ faults}{All\ normal\ operation\ samples} \quad (22)$$

The second metric is the **Missed Detection Rate (MDR)**, which represents the percentage of the fault samples that were not detected, as shown in Equation (23).

$$MDR = \frac{FN}{TP + FN} = \frac{Undetected\ fault\ samples}{All\ fault\ samples} \quad (23)$$

The third and final metric, **Time to Detection (TTD)**, is not determined by concepts from the confusion matrix, but by the time it takes for the model to predict the fault once the fault has started. For this metric, for better handling false alarms, fault detection is only considered after six consecutive predicted fault events (RUSSELL; CHIANG; BRAATZ, 2000). For f as the timestep in which the fault starts and d as the first timestep of the six consecutive detections, TTD is simply defined as shown in Equation (24).

$$TTD = (d - f) \times 1\ observations = (d - f) \times 3\ minutes \quad (24)$$

For the RL framework, multiple instances of the custom RL environment were created, each of them with a different simulation seed, so that, for each run, while the general dynamics of the chemical process were the same, the output would be different in terms of initialization parameters and noise. This allows the agent's generalization capacity to be tested in a similar way as the testing of supervised learning algorithms, in which models have to be tested with unseen test datasets to ensure that the learning process was really effective and the methods will work in future data, without problems such as overfitting. Hence, the framework uses both **training** and **testing** environments. Also, a **validation environment** was created, and, for every 100 training episodes, the current trained model was validated, in order to keep track of the actual model performance in parallel with

the training process. For both validation and training environments, the fault start up is according to the default CE design, starting in a random time step, and the simulation duration was 24h. However, for the test environment, a slight alteration was made based on the work of Russell, et al. (2000) for comparison purposes, in which the test data had 48h simulations (960 observations) with the faults starting after the first 8h, or 160 observations.

It is crucial to notice that, while the agent is always under the ϵ-greedy policy exploration method while in the training environment, with 0.05 as the minimum value of ϵ, this is not true for the validation and testing environments, and for those the agent always picks the action with the highest probability. Additionally, for each validation step performed, the algorithm checks whether or not the current total reward is the greatest so far and, if so, saves the model parameters from this validation step. This allows the model with the best validation performance to be saved and compared with the final output model of the training process. The best overall model is then selected as the final model and used in the testing environment to generate the performance metrics, which are obtained from the mean value of each metric acquired from all testing episodes.

Additionally, as mentioned before in section 3.2.2. Principal Component Analysis (PCA), a PCA model was developed as a baseline for comparison purposes with the RL results. The PCA was built in terms of the normal operation data (no faults present) and the detection was performed using two different tests, using both T^2 and the Q statistics, with respective thresholds (upper control limit) being set as the 99% percentile (that is, false alarm rate of 1%). This means that, for a dataset transformed by the PCA model's loading vectors, the observations which yield values greater than the respective thresholds of the statistics are classified as faults in the test. **Figure 15** shows the complete flowchart for the training, validation and testing steps, including the PCA model and the performance metrics analysis.

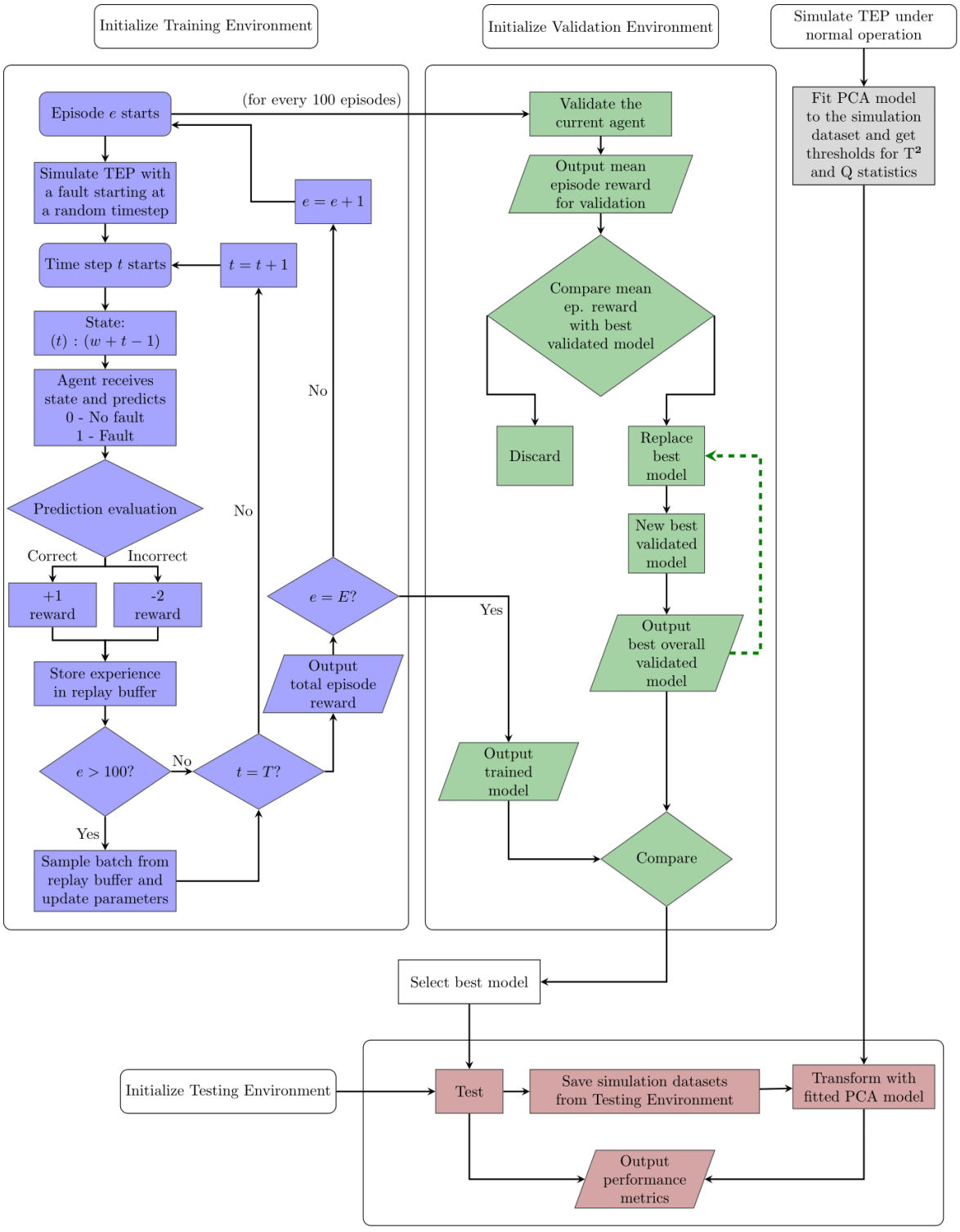


Figure 15: Complete flowchart of the proposed framework for training, validating and testing of RL agent for performing fault detection in a continuous chemical process.

The process described in Figure 15 was performed, for each architecture, a total of twenty times, one for each of the selected faults. The training process for each fault was set to a total of 3000 training episodes, and each validation step consisted of 600 episodes. For the testing step, 1000 episodes were performed, and the datasets for the test simulations were saved and then applied to the

baseline PCA model. The results are discussed in the next section. A particular and important difference between the proposed framework and the baseline method is that for the RL framework the input data will be the sliding window, and therefore there will be an overlap between two consecutive states, and however for the PCA model each state will be one individual observation.

6. RESULTS AND DISCUSSION

According to the methodology presented in section 5, two RL agents, one for each of the proposed DQN MLP architectures, defined after some trials, were built for each of the faults selected, meaning a total of 40 agents. This current chapter is divided into subsections, dedicated to the results of the PCA model as well as the results for the RL models. As mentioned in the previous section, all models were trained for 3000 episodes. In parallel, after every 100 training episodes, 600-episode validation steps were performed in order to keep track of the performance without the ϵ-greedy policy, which is applied during training. Also, the validation steps were used to keep track of the best model in terms of mean episode rewards, which allowed the best overall model to be selected as the final output RL agent. This agent was then tested with 1000 fault-detection episodes. For training, validation and testing, different environments were used in order to guarantee different seeds and to avoid overfitting and promoting generalization ability for the agents. For simplification purposes, the (260,256,128,64,2) architecture will be called **Architecture 1 (or Arch1)**, and the (260,64,128,64,2) architecture will be called **Architecture 2 (or Arch2)**.

Additionally, a fault detection performance analysis based on grouping the twenty faults of the TEP benchmark according to the difficulty of detection (GUTIERREZ-ROJAS et al., 2022) was also conducted. Such grouping was based on the PCA results for the T^2 and Q statistics, which were also used in the present work for comparison purposes. Namely, group 1: easy-to-detect faults (1, 2, 4, 6, 7, 8, 12, 13, 14); group 2: intermediate-detection faults (11, 17, 18); group 3: hard-to-detect faults (5, 10, 16, 19, 20); and group 4: very hard-to-detect faults (3, 9, 15). Finally, a model complexity analysis was performed in order to verify the relationship between model complexity and fault detection performance, with Arch1 having a relatively larger number of parameters than Arch2, as previously mentioned.

6.1. PCA BASELINE

The PCA model was created and fitted to a thousand 24h simulations under normal operation. For the new latent variables, the principal components, the first aspect analyzed was the explained variance. The goal was to build a baseline model with 80% explained variance and, for this particular PCA model, 25 principal components were retained to achieve it, as shown in **Figure 16**. Therefore, to build the PCA-based fault detection model, 25 components were considered in order to obtain both T^2 and Q statistics, as well as the respective thresholds (or control limits) for both statistics.

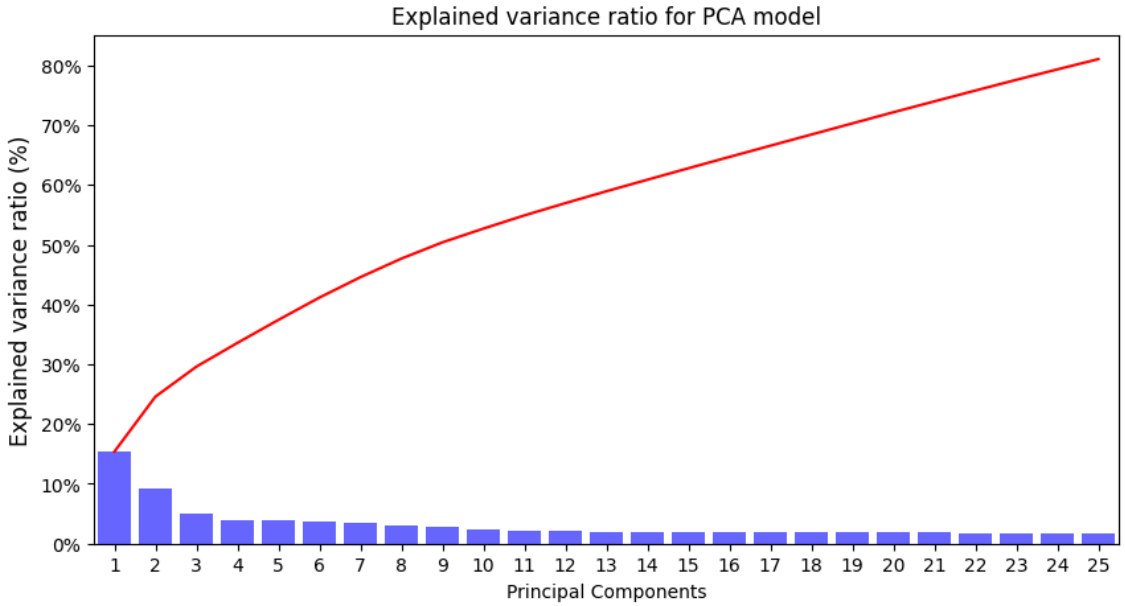


Figure 16: Explained variance for the first 25 components of the PCA baseline model.

Then, with the normal operation-based model, it was possible to use its loading vectors to transform the data obtained from the same simulations generated by the test environment during the testing of the RL agents for each of the twenty faults. The upper thresholds for the T^2 and Q statistics were obtained from the 99% percentile of the values of both statistics from the PCA model, considering 25 components, from a second dataset relative to normal operating conditions. For T^2, the threshold was 44.3010, and for Q, the control limit was 19.8975. With these values, the simulation sets for all faults were scaled (zero mean and unit variance) according to the normal operation data and transformed by the PCA model and the T^2 and Q statistics were calculated using the test dataset. For both statistics, if the statistic value from a specific observation exceeded the threshold, a fault is detected and, otherwise, it is recognized as a normal operation sample. **Table 3** contains the performance metrics for all twenty faults.

Table 3 – Performance metrics (for FAR and MDR, in %; for TTD, in average number of observations) of the PCA model (T^2 and Q) in the test step. The absence of a value for TTD means that the six sequential detections did not happen for the particular fault. The values in bold represent the best result for each metric, for each fault.

Fault	FAR		MDR		TTD	
	T^2	Q	T^2	Q	T^2	Q
1	**0.67**	0.89	0.71	**0.42**	9.91	**7.50**
2	**0.74**	0.87	**1.48**	1.62	**16.52**	18.40
3	**0.75**	0.93	**98.80**	98.91	-	-
4	**0.70**	0.91	89.54	**0.17**	95.45	**5.40**
5	**0.69**	0.86	**76.08**	87.99	10.99	**6.10**
6	**0.67**	0.94	1.05	**0.12**	12.81	**5.00**

7	**0.72**	0.92	**0.12**	0.13	**4.99**	5.00
8	**0.70**	0.90	**3.16**	7.73	**27.95**	31.80
9	**0.69**	0.90	**98.75**	98.89	-	-
10	**0.67**	0.90	**79.15**	79.47	233.63	**187.40**
11	**0.67**	0.88	67.96	**23.94**	109.45	**15.30**
12	**0.68**	0.89	**2.36**	5.17	17.64	**15.80**
13	**0.76**	0.91	**6.30**	6.34	55.35	**49.40**
14	**0.66**	0.93	3.34	**0.17**	7.85	**5.30**
15	**0.77**	0.93	**98.57**	98.82	-	-
16	**0.66**	0.88	92.60	**86.02**	262.62	**244.00**
17	**0.71**	0.90	28.66	**9.21**	44.84	**37.90**
18	**0.69**	0.90	7.63	**6.77**	67.18	**59.60**
19	**0.67**	0.94	**88.89**	93.24	**59.13**	-
20	**0.71**	0.89	76.46	**51.21**	81.81	**50.80**

From the table above, we can see that, as expected from the definition of the threshold as the 99% percentile for both statistics, the FAR values were around 1% for all faults. For MDR, for the T^2 statistic, the PCA model presented values lower than 1% for faults 1 and 7 (group 1, easy detection), values between 1% and 5% for faults 2, 6, 8, 12 and 14 (also from group 1), and values between 5% and 10% for faults 13 (group 1) and 18 (group 2, intermediate detection). The highest MDR results were obtained for the very hard-to-detect (group 4) faults (3, 9 and 15) with rates over 98%. The hard-to-detect faults (group 3) presented MDR values of at least 76.08% (fault 5). For TTD, the lowest detection delays were presented for faults 7 (4.99 time units), 14 (7.85 time units), and 1 (9.91 time units). The remaining faults took at least 10 observations, i.e., 30 minutes (given the 3-minute sampling frequency) to be detected. Among them, it is important to notice that six consecutive detections did not happen for group 4, and thus this metric could not be computed.

For the Q statistic test, the MDR results were lower than 1% for faults 1, 4, 6, 7, and 14 (group 1), and between 1% and 5% for fault 2, also easy-to-detect. Faults 8, 12, 13 (group 1), 17, and 18 (group 2) presented rates between 5% and 10%. The very hard-to-detect group presented the highest rates (over 98%), as it was the case for the T^2 statistic, and the hard-to-detect faults were missed at a rate of at least 51.21% (fault 20). Regarding TTD, the minimum requirements for detection weren't matched for faults 3, 9, 15 (group 4), and 19 (group 3), so the metric was not computed. The lowest detection delays resulted from faults 6, 7, 14, 4, 5 and 1, with respective TTD values of 5.00, 5.00, 5.30, 5.40, 6.10 and 7.50 observations. The remaining faults took at least 15.30 observations (45.90 minutes) to be detected.

The overall performance of the PCA baseline was satisfactory for group 1 (easy-to-detect faults), but even some of these faults presented high TTD results, e.g. fault 8, with 27.95 observations, or 83.85 minutes. For group 2, the performance was already significantly worse, and this pattern carried on for the increasingly detection difficulty, as expected and shown in **Table 4**. Also, the

results for the Q test, on average, were noticeably better than those for the T^2 test, with lower average MDR and TTD values for all groups.

Table 4 - Performance metrics (for FAR and MDR, in %; for TTD, in average number of observations) of the PCA model (T^2 and Q) for different levels of detectability. Values in parenthesis represent standard deviations. The absence of a value for TTD means that the six sequential detections did not happen for the particular group. The values in bold represent the best result for each metric, for each group.

Group (difficulty)	FAR		MDR		TTD	
	T^2	Q	T^2	Q	T^2	Q
1 (Easy)	**0.70 (0.03)**	0.91 (0.02)	12.01 (29.13)	**2.43 (3.09)**	27.61 (29.64)	**15.96 (15.46)**
2 (Intermediate)	**0.69 (0.02)**	0.89 (0.01)	34.75 (30.62)	**13.31 (9.29)**	73.82 (32.81)	**37.60 (22.15)**
3 (Hard)	**0.68 (1.50)**	0.89 (0.03)	82.64 (7.61)	**79.59 (16.61)**	129.64 (111.62)	**122.08 (112.05)**
4 (Very hard)	**0.74 (0.04)**	0.92 (0.02)	**98.71 (0.12)**	98.87 (0.05)	-	-

6.2. DEEP REINFORCEMENT LEARNING AGENTS

As described previously, two agents, with different DQN-MLP architectures (Arch1 and Arch2), were obtained for each of the twenty faults using the RL framework, resulting in a total of forty different agents trained, validated and tested for fault detection in the TEP benchmark. The results of both architectures were used to verify the relationship between model complexity and fault detection performance, with Arch1 being more complex (relatively larger number of parameters) than Arch2. **Figure 17** shows representative learning curves for four faults (2, 5, 11 and 15) for the Arch2-based agent. The remaining learning curves for both architectures are in the Appendix. When analyzing the behavior of these curves, it is possible to observe different behaviors for each fault. From the definition of the reward function, which yielded a reward of -2 for incorrect predictions and +1 for correct predictions, it was expected that the average reward for 480 timesteps would be around -240 in the beginning of the training phase, where randomly initialized models have a 50% chance of getting correct predictions, and this is noticeable for all four faults.

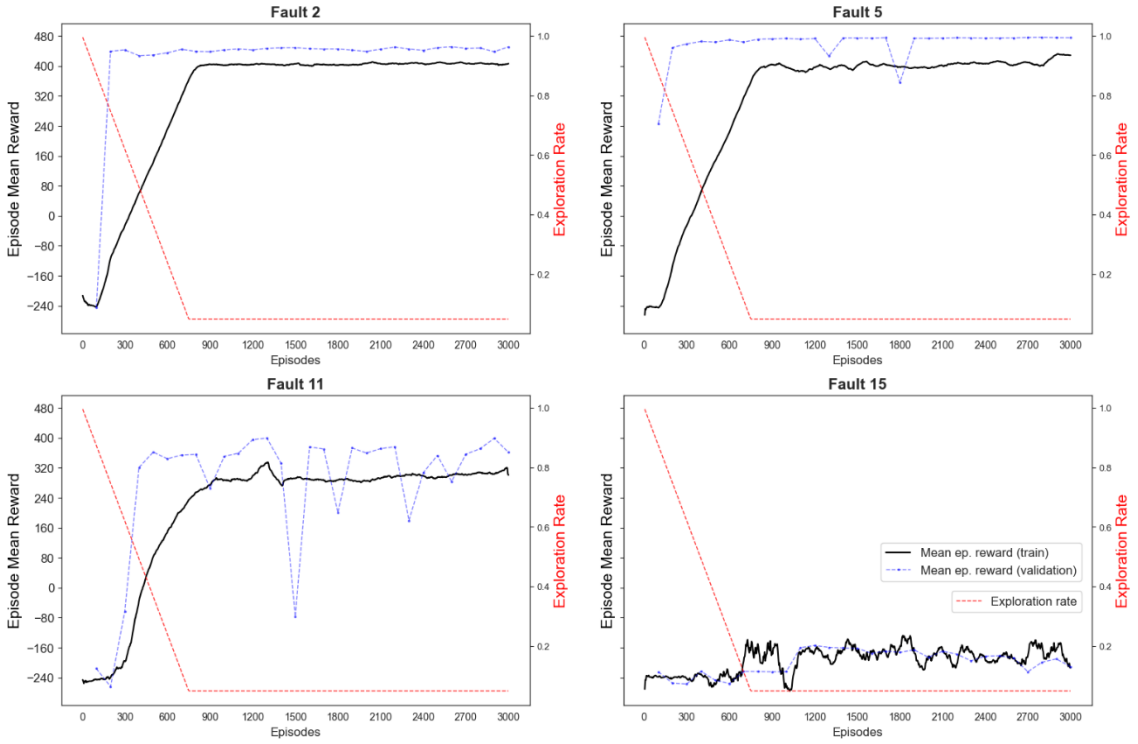


Figure 17: Training and validation curves (mean episode reward) during the training phase of the Arch2-based RL agents for faults 2, 5, 11 and 15.

Furthermore, it can be observed that the learning process was smoother for faults 2 and 5 than it was for faults 11 and 15. When comparing the training and validation curves for these faults, it is possible to notice, due to the ϵ-greedy policy, which guarantees a minimum exploration rate of 5% during training, that the validation rewards improved faster and were maintained higher than the training rewards for all faults, with an exception for fault 15, which has a very low known degree of detectability (RUSSELL; CHIANG; BRAATZ, 2000). For the latter, the average reward remained negative during training, and there is also no observable correlation between the decrease in the exploration rate and increase in performance over time, which was true for the other three faults selected. As expected, this initial qualitative analysis indicates different fault performances and training behaviors depending on the fault type.

According to the proposed framework, for each of the architectures, the best model in terms of the mean rewards for the validation steps was selected as the final trained model and then tested to obtain the FAR, MDR, and TTD metrics. Each performance metric refers to the mean value from the 1000 test episodes. The results are presented in **Table 5**, **Table 6** and **Table 7**, together with the baseline PCA results for both T^2 and Q statistics. For **Table 6** and **Table 7**, there are values for absolute gain between the RL-based approach and both PCA statistics for each fault. This gain is given by the difference between the best RL result (considering Arch1 and Arch2) and those of T^2 and Q. The more positive,

the better the RL-based result. For all metrics, the best result for each fault is highlighted in bold.

For this analysis, to better understand and evaluate model performance, it is crucial that all metrics are analyzed together. Focusing on a single metric, such as a low MDR, can be misleading. For example, a low MDR might seem indicative of good performance, but if accompanied by a high FAR, it suggests the model frequently signals false faults, reducing its effectiveness. Similarly, a model with a low FAR but high MDR overlooks many actual faults, making it unreliable. The significance of TTD also cannot be overstated; while quick detection is advantageous, it should not compromise the accuracy reflected in FAR by outputting frequent false alarms. This can be seen in **Table 3**, where the PCA models for all faults have FAR values lower than 1%, but also output MDR values close to 100%, which is the case for faults 3, 9 and 15 (with over 98% for both Q and T^2 statistics).

In the present analysis, a FAR value (**Table 5**) is considered satisfactory if it is less than 5%. The values above are identified by gray cells and, if it happens, then the particular model will not be considered valid, due to the aforementioned trade-off. For this metric, both RL agents obtained satisfactory values, with an exception for faults 3, 9 and 15 (which are precisely the faults from group 4), whose very high detection difficulty is well known (RUSSELL; CHIANG; BRAATZ, 2000). The lowest value in this case was 27.22% for fault 9 and Arch2, which is considerably high. The other exception refers to Arch1 and fault 19, a hard-to-detect fault (group 3), with a FAR value of 8.95%, which is a magnitude often encountered in the process industry, but that also can discredit the monitoring system. In contrast, Arch2 yielded a satisfactory FAR of 3.76% for the same fault, and this agent will be considered for further analysis. Finally, the RL-based approach presented lower FAR values for thirteen of the twenty faults compared to PCA and, in general, as desired, the values are less than 1% for both RL and PCA approaches.

Table 5 – FAR metric (in %) for the RL and PCA approaches (Gray cells mean FAR > 5%; bold values are the best overall result for each fault; and "-" means undetected fault).

Fault	RL (Arch1)	RL (Arch2)	PCA (T^2)	PCA Q
1	**0.00**	0.13	0.67	0.89
2	**0.14**	0.17	0.74	0.87
3	28.79	35.51	**0.75**	0.93
4	**0.01**	0.02	0.70	0.91
5	**0.04**	0.17	0.69	0.86
6	**0.00**	**0.00**	0.67	0.94
7	**0.00**	**0.00**	0.72	0.92
8	**0.11**	0.13	0.70	0.90
9	40.16	27.22	**0.69**	0.90
10	2.34	2.12	**0.67**	0.90
11	1.32	**0.06**	0.67	0.88
12	**0.04**	0.18	0.68	0.89

13	**0.37**	0.71	0.76	0.91
14	0.48	**0.00**	0.60	0.93
15	45.43	54.65	**0.70**	0.93
16	0.27	**0.25**	0.66	0.88
17	1.31	3.55	**0.71**	0.90
18	0.10	**0.03**	0.69	0.90
19	8.95	3.76	**0.67**	0.94
20	1.58	2.03	**0.71**	0.89

For the MDR metric (**Table 6**), significant RL-based gains were obtained. The main gains resulted from fault 5 (75.88% and 87.79% in relation to T^2 and Q, respectively), fault 10, (67.64% and 67.96%), fault 16 (88.48% and 81.90%), fault 19 (53.89% and 58.24%) and fault 20 (61.42% and 36.17%). This set comprises precisely all hard-to-detect faults (group 3). For example, the MDR for fault 5 dropped significantly from 76.08% and 87.99% for T^2 and Q, respectively, to 0.22% and 0.20%, for Arch1 and Arch2, respectively. Therefore, it can be noted that this initially hard-to-detect fault has become satisfactorily detectable. The same can be observed for faults 10, 16, and 20, with maximum MDR values of 15%, and fault 19, to a lesser extent, with a MDR of 35.00%. The maximum FAR value considering all these faults was equal to 3.76% for fault 19 and Arch2 (**Table 5**). There were also significant gains for the fault 11, more specifically 63.38% and 19.36% for T^2 and Q, respectively. This fault belongs to group 2, i.e., has an intermediate level of detection difficulty. In the PCA model, the MDR values for this fault were 67.96% for T^2 and 23.94% for Q, as opposed to 7.34% and 4.58%, for the Arch1-based and Arch2-based models, respectively. Other notable gains were obtained for fault 4, an easy-to-detect fault (89.42% in relation to T^2), and fault 17, of intermediate detectability (22.73%, also for T^2). Moreover, the RL-based results were at least similar to the PCA-based results for all faults, which means the significant gains obtained for one set of faults did not come at the expense of deteriorating detection performance in other faults. The only faults in which the PCA model reached lower missed detection rates were faults 6 (-0.01% for Q), fault 7 (-0.01% for T^2), and fault 14 (-0.11% for Q). Nevertheless, the difference is noticeably significantly small, and RL and PCA-based approaches achieve similar results, with very low MDR values.

Table 6 – MDR metric (in %) for the RL and PCA approaches (Gray cells mean FAR > 5%; bold values are the best overall result for each fault; and "-" means undetected fault).

Fault	RL (Arch1)	RL (Arch2)	PCA (T^2)	PCA (Q)	Absolute gain (T^2)	Absolute gain (Q)
1	0.38	**0.31**	0.71	0.42	0.40	0.11
2	**1.14**	1.28	1.48	1.62	0.34	0.48
3	53.31	43.31	**98.80**	98.91	-	-
4	0.12	**0.12**	89.54	0.17	89.42	0.05
5	0.22	**0.20**	76.08	87.99	75.88	87.79
6	0.13	0.13	1.05	**0.12**	0.92	-0.01
7	0.13	0.13	**0.12**	0.13	-0.01	0.00
8	2.56	**2.53**	3.16	7.73	0.63	5.20

9	38.55	52.24	**98.75**	98.89	-	-
10	11.94	**11.51**	79.15	79.47	67.64	67.96
11	7.34	**4.58**	67.96	23.94	63.38	19.36
12	**0.04**	1.45	2.36	5.17	2.32	5.13
13	**4.72**	4.95	6.30	6.35	1.58	1.62
14	6.37	0.28	3.34	**0.17**	3.06	-0.11
15	34.35	27.41	**98.57**	98.82	-	-
16	**4.12**	4.15	92.60	86.02	88.48	81.90
17	8.79	**5.93**	28.66	9.21	22.73	3.28
18	**5.96**	6.10	7.63	6.77	1.67	0.81
19	31.12	**35.00**	88.89	93.24	53.89	58.24
20	**15.04**	15.36	76.46	51.21	61.42	36.17

The TTD metric (**Table 7**) refers to the average detection delay in terms of number of observations or time units (the sampling frequency for the TEP benchmark is three minutes). As it was the case for the MDR metric, the TTD values for hard-to-detect faults (group 3) presented significant improvement. The main gains refer to fault 10 (233.63 and 187.40 time units in relation to T^2 and Q, respectively), and fault 16 (252.34 and 233.72 time units). To a lesser extent, fault 5 presented gains of 9.37 and 4.48, respectively, and, for fault 20, 47.42 and 16.41 observations. A particular case concerns fault 19, which is only detectable by the T^2 threshold (59.13 time units) for the PCA baseline approach, whereas the corresponding value for the Arch2-based model was more than five times smaller (10.43 observations). These results, coupled with the previous results for MDR and FAR, illustrate the effectiveness of the RL-based approaches for improving detection for group 3 (hard-to-detect faults) in the TEP benchmark. Other notable gains in TTD resulted from fault 4 (94.45 time units in relation to T^2), fault 8 (10.15 and 14.00 time units in relation to T^2 and Q, respectively), fault 13 (19.90 and 13.95 time units), and fault 18 (19.81 and 12.23 time units). Overall, the TTD results were lower for all valid (FAR > 5%) RL models.

Table 7 - TTD metric (in average number of observations, or time units) for the RL and PCA approaches (Gray cells mean FAR > 5%; bold values are the best overall result for each fault; and "-" means undetected fault).

Fault	RL (Arch1)	RL (Arch2)	PCA (T^2)	PCA (Q)	Absolute gain (T^2)	Absolute gain (Q)
1	3.06	**2.47**	9.91	7.50	7.44	5.03
2	**9.18**	10.32	16.52	18.40	7.34	9.22
3	25.94	14.91	-	-	-	-
4	**1.00**	**1.00**	95.45	5.40	94.45	4.40
5	1.75	**1.62**	10.99	6.10	9.37	4.48
6	1.02	**1.01**	12.81	5.00	11.80	3.99
7	**1.00**	**1.00**	4.99	5.00	3.99	4.00
8	18.21	**17.80**	27.95	31.80	10.15	14.00
9	12.41	23.65	-	-	-	-
10	26.51	**0.00**	233.63	187.40	233.63	187.40
11	6.49	**6.31**	109.45	15.30	103.14	8.99
12	9.29	**8.71**	17.64	15.80	8.93	7.09
13	36.72	35.45	55.35	49.40	19.90	13.95

14	5.00	**2.49**	7.85	5.30	5.36	2.81
15	9.31	5.84	-	-	-	-
16	**10.28**	**10.28**	262.62	244.00	252.34	233.72
17	33.77	**32.36**	44.84	37.90	12.48	5.54
18	**47.37**	48.82	67.18	59.60	19.81	12.23
19	9.27	**10.43**	59.13	-	49.86	-
20	34.50	**34.39**	81.81	50.80	47.42	16.41

An important point to keep in mind is the impact of the sliding window length, given the continuous nature of the process operations. As mentioned previously, a single fault event among the observations that make up the time window causes its true label to be set as fault. This means that transition periods from normal operation to a fault condition are labeled as fault, even when most observations refer to normal operation. Such a definition might impact performance metrics, especially MDR and TTD. In this work, a window size of 5 was adopted in order to balance these metrics with the window length.

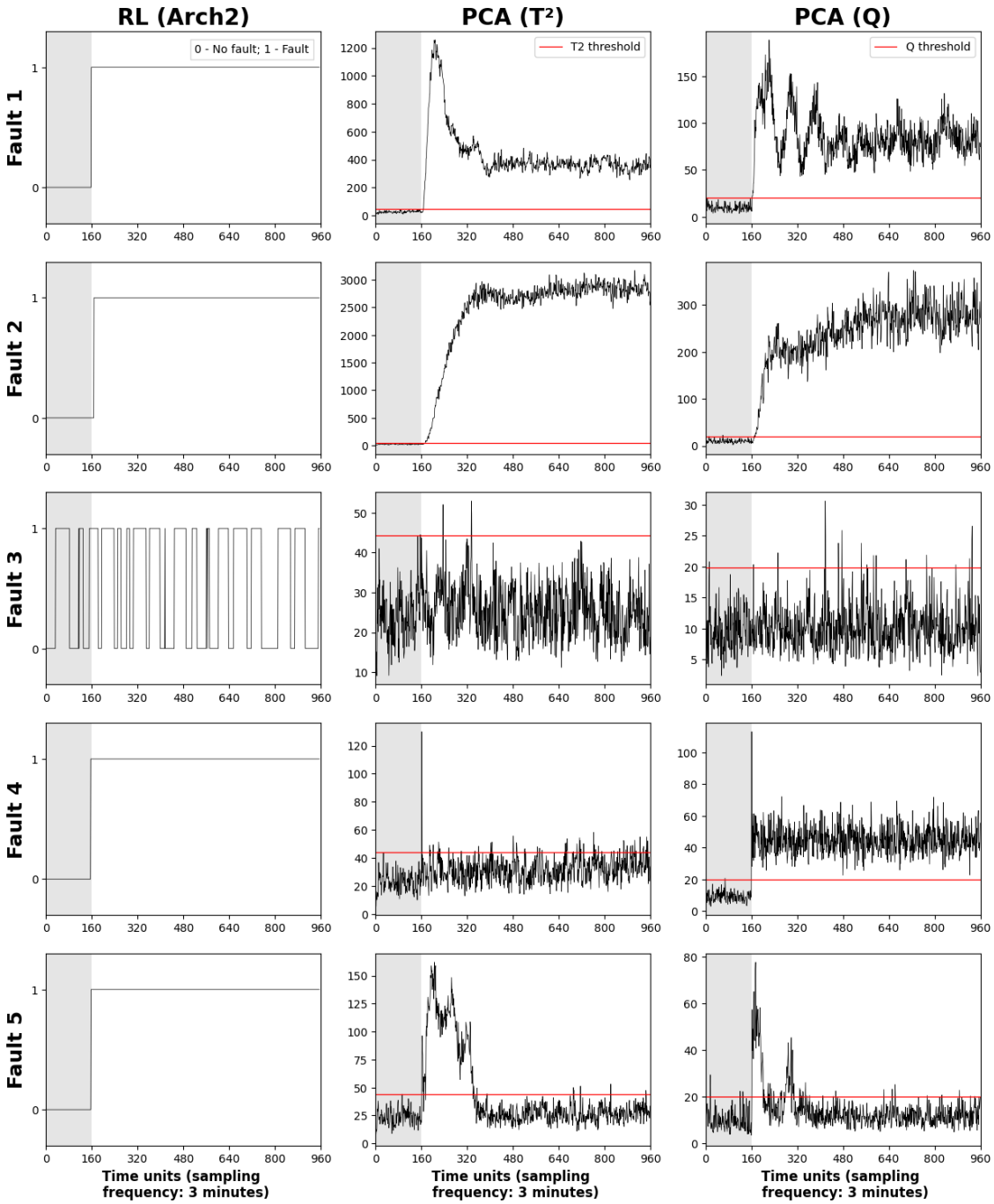


Figure 18, **Figure 19**, **Figure 20** and **Figure 21** show detection plots for the Arch2-based agents for the twenty faults in the test set. As mentioned in Section 5, during the test set the faults always start in the 160th observation (corresponding to 8h of process simulation). The first column in each figure refers to the RL-model, with 0 meaning normal operation and 1, a fault, in the y-axis. For the other two columns, the plots refer respectively to T^2 and Q statistics. These plots illustrate the detection process for the RL and the PCA baseline approaches, with some faults being easily detectable by both approaches, as is the case for fault 1; some are easily detectable by the RL agents but not by the PCA models, e.g., fault 5; and some are not satisfactorily detected by any of the models, e.g., fault 3. The detection plots for Arch1 are shown in the Appendix.

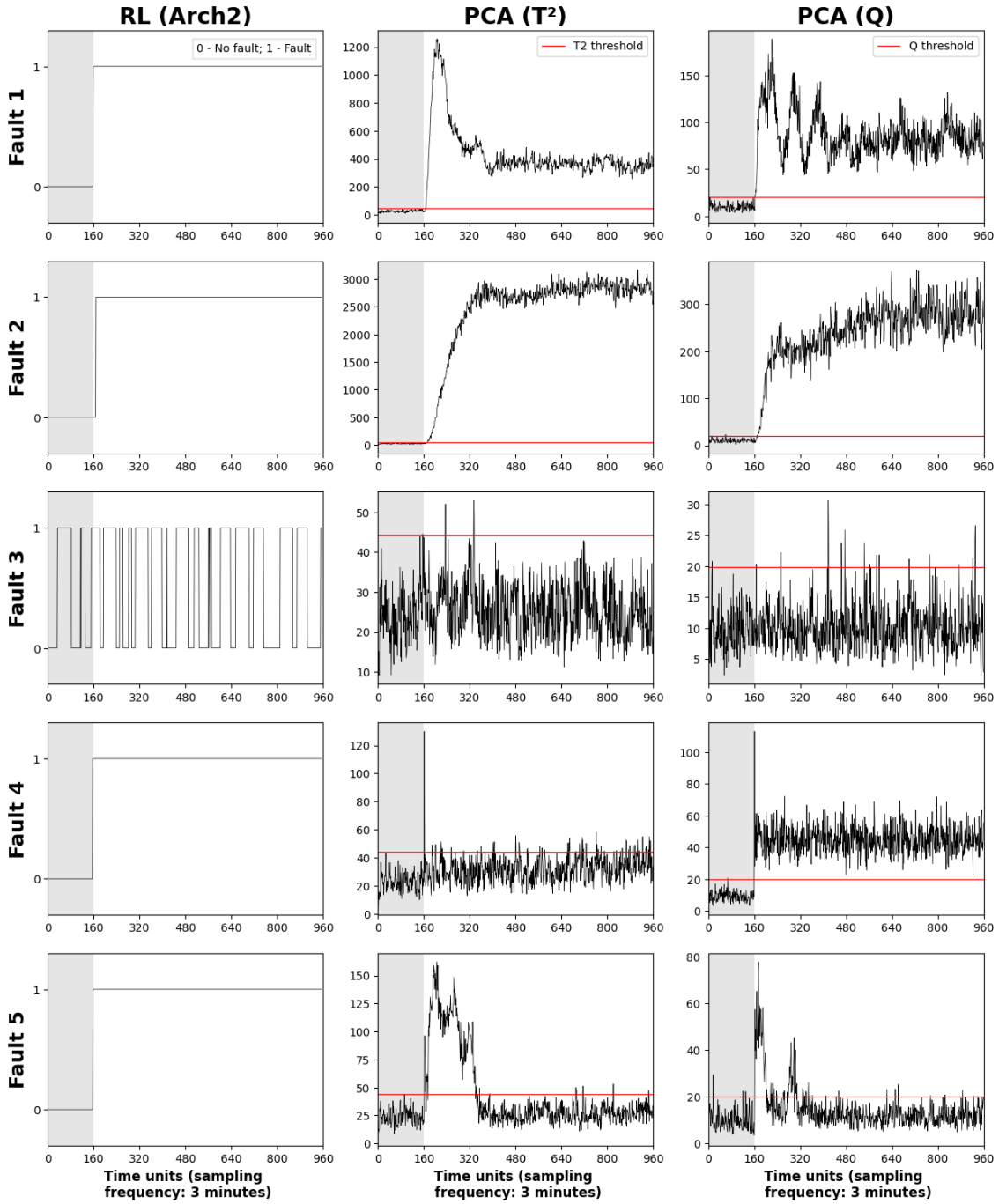


Figure 18: Detection plots (faults 1 to 5) for the RL-based model (Arch2), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.

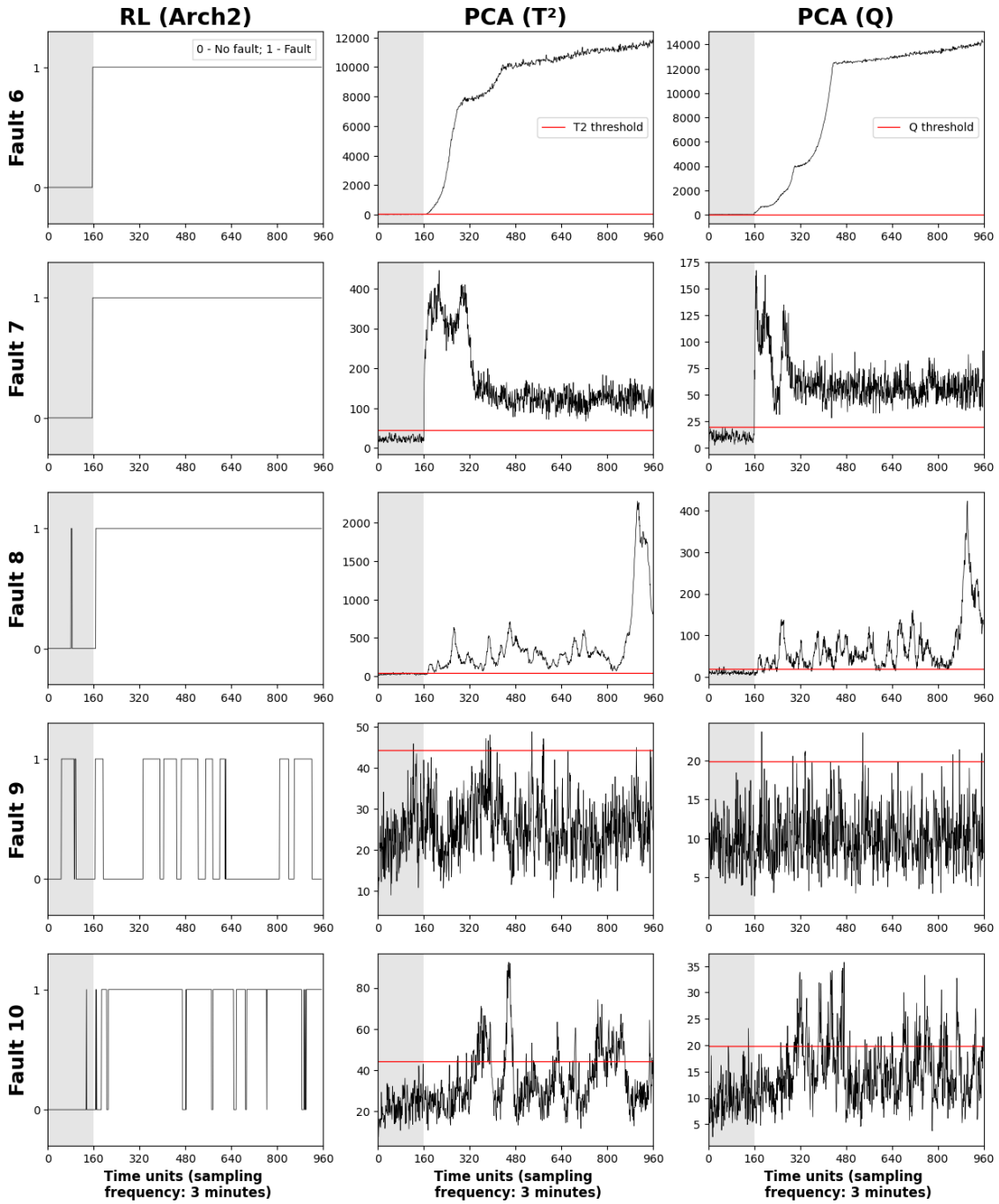


Figure 19: Detection plots (faults 6 to 10) for the RL-based model (Arch2), the T² test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.

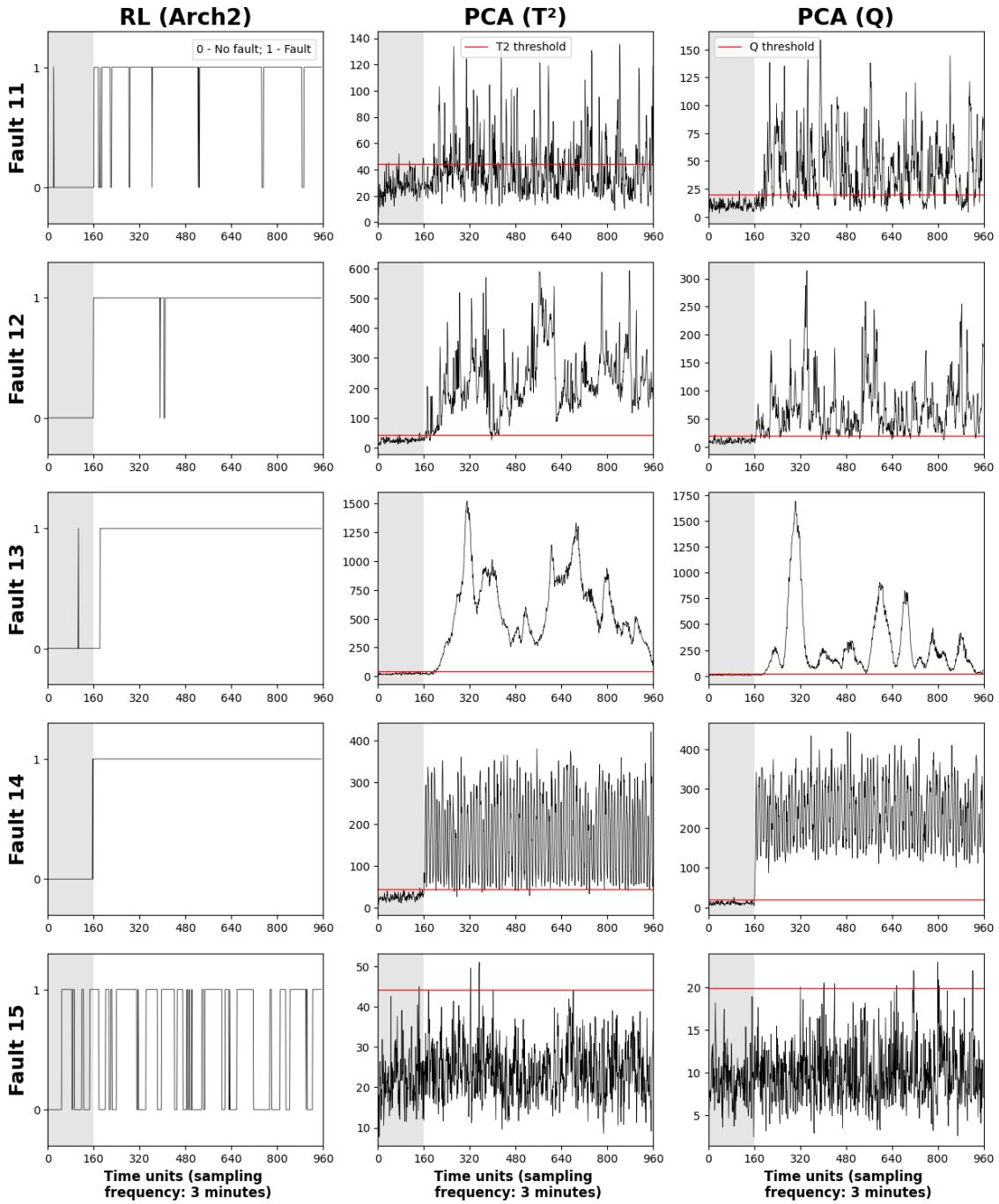
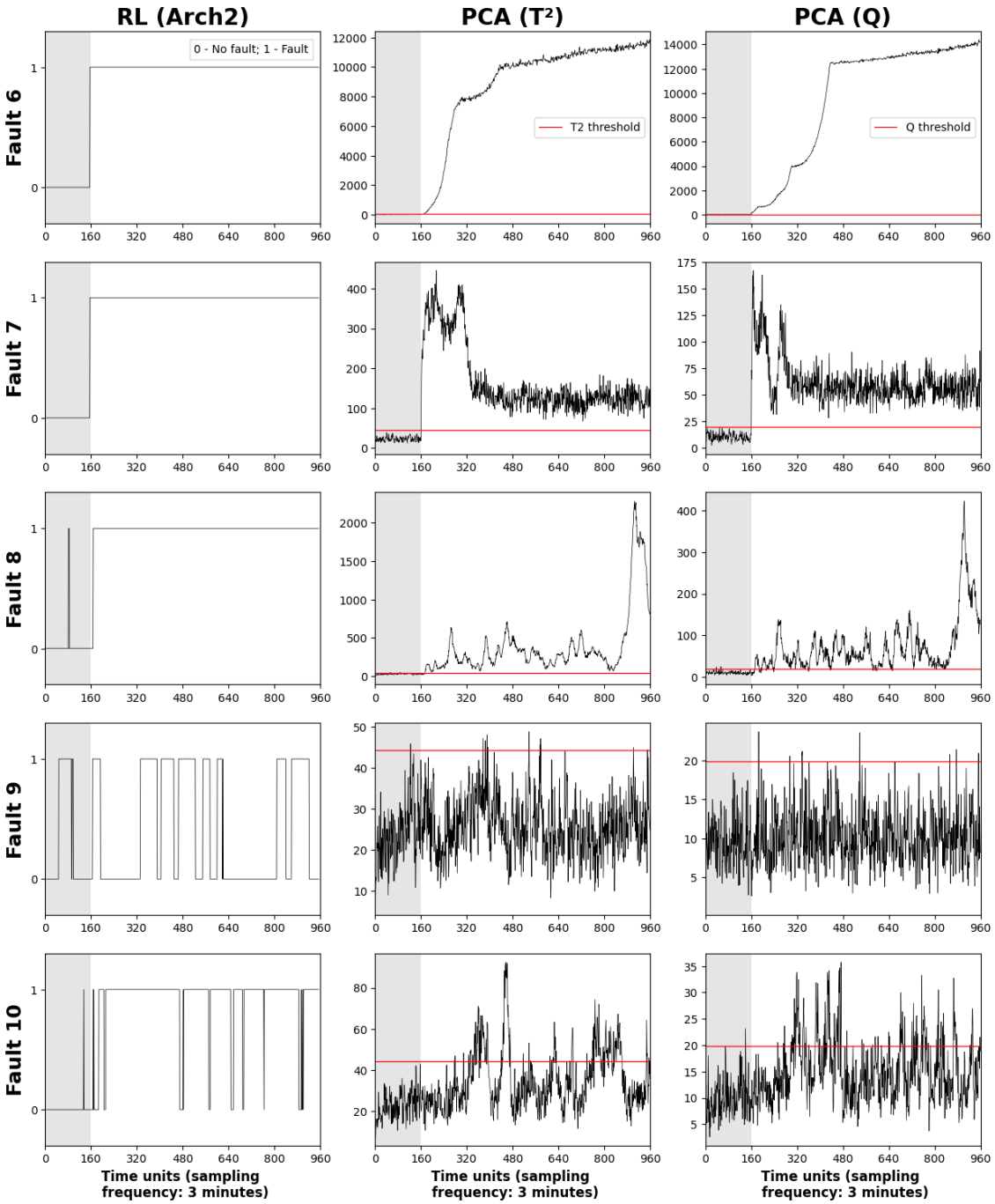


Figure 20: Detection plots (faults 11 to 15) for the RL-based model (Arch2), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.


RL (Arch2)
PCA (T²)
PCA (Q)
Fault 6
Fault 7
Fault 8
Fault 9
Fault 10
0 - No fault; 1 - Fault
T2 threshold
Q threshold
Time units (sampling frequency: 3 minutes)
Time units (sampling frequency: 3 minutes)
Time units (sampling frequency: 3 minutes)

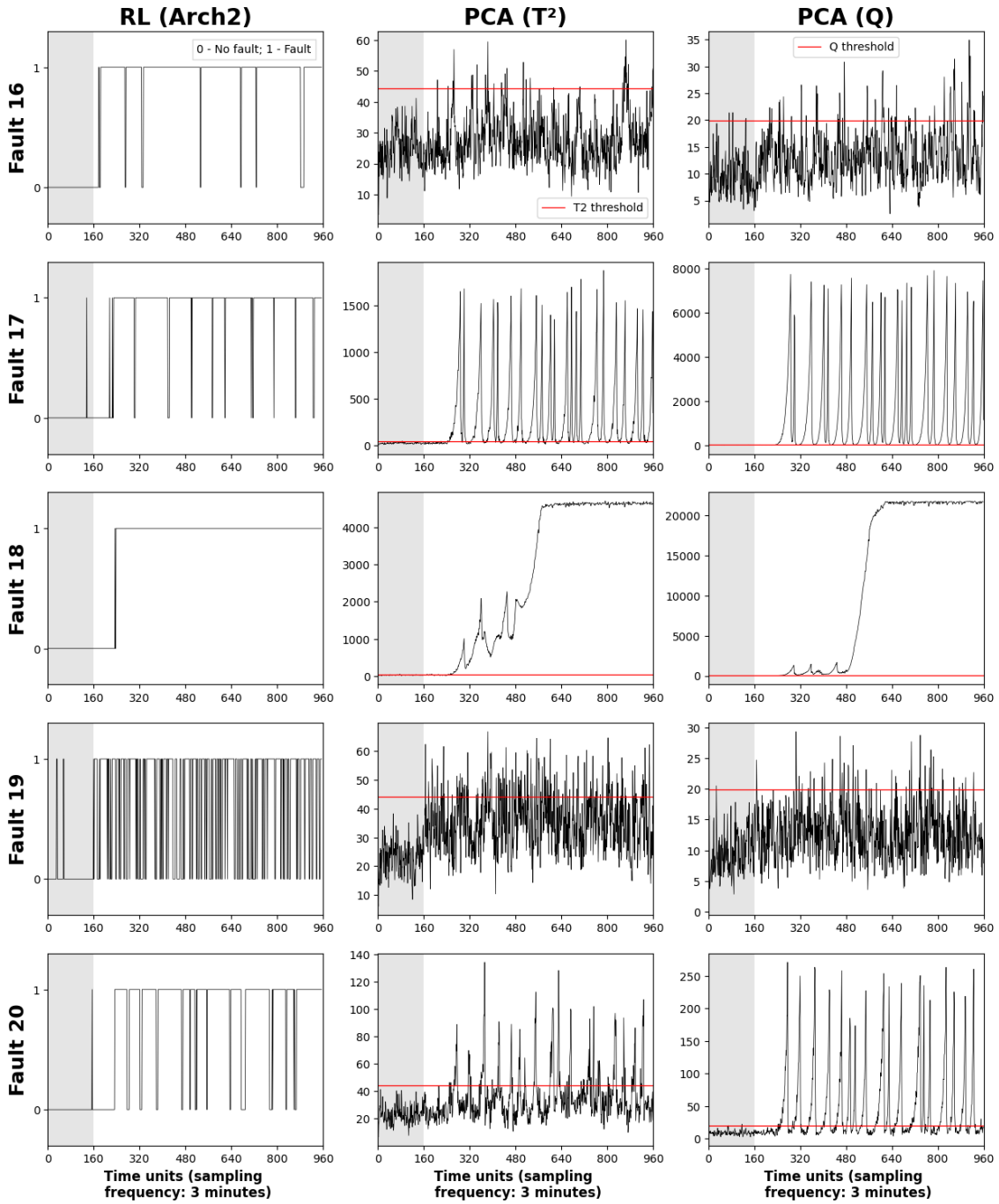


Figure 21: Detection plots (faults 16 to 20) for the RL-based model (Arch2), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.

6.3. DETECTABILITY ANALYSIS

In addition to the analysis on the previous section, another analysis was carried out separately on the detection capacity of the RL and PCA approaches. **Table 8**, **Table 9** and **Table 10** contain average values by group of faults for FAR, MDR and TTD. The corresponding standard deviation values are shown in parentheses. For FAR (**Table 8**), the average values for the RL approach increase smoothly with the group's detection difficulty. The exception was group 4, i.e., the very hard-to-detect faults. For easy-, intermediate- and hard-to-detect faults (groups 1, 2 and 3, respectively), the false alarm rates were maintained lower than 5%, the desired criteria for valid models. Due to the nature of both statistical thresholds for the PCA model, the FAR values for the baseline are more stable and closer to 1% (especially for the Q statistic).

Table 8 – Average FAR values (in %) for the four fault groups. Values in parenthesis represent standard deviations (Gray cells mean FAR > 5% and bold values the best overall result given the fault group).

Group (difficulty)	RL (Arch1)	RL (Arch2)	PCA (T^2)	PCA (Q)
1 (Easy)	**0.13 (0.18)**	0.15 (0.22)	0.70 (0.03)	0.91 (0.02)
2 (Intermediate)	0.91 (0.70)	1.21 (2.02)	**0.69 (0.02)**	0.89 (0.01)
3 (Hard)	2.64 (3.65)	1.67 (1.50)	**0.68 (1.50)**	0.89 (0.01)
4 (Very-hard)	38.13 (8.50)	39.13 (14.07)	**0.74 (0.04)**	0.89 (0.01)

The MDR values in **Table 9** show that, for both RL and PCA approaches, the missed detection rates increase with detection difficulty. For the RL approaches, the average MDR values are considerably lower. For hard-to-detect faults (group 3), the average values are 12.49% (Arch1) and 13.24 (Arch2), which, when compared to the values for the T^2 and Q statistics, respectively 82.64% and 79.59%, can be denoted as significantly low.

Table 9 – Average MDR values (in %) for the four fault groups. Values in parenthesis represent standard deviations (Gray cells mean FAR > 5% and bold values the best overall result given the fault group).

Group (difficulty)	RL (Arch1)	RL (Arch2)	PCA (T^2)	PCA (Q)
1 (Easy)	1.73 (2.34)	**1.24 (1.62)**	12.01 (29.13)	2.43 (3.09)
2 (Intermediate)	7.36 (1.42)	**5.54 (0.83)**	34.75 (30.62)	13.31 (9.29)
3 (Hard)	**12.49 (11.98)**	13.24 (13.54)	82.64 (7.61)	79.59 (16.61)
4 (Very-hard)	42.07 (9.96)	40.99 (12.58)	**98.71 (0.12)**	98.87 (0.05)

Regarding TTD, the behavior was different than it was for the previous metrics, and such smooth relation between the detection delays and the difficulty level was not observed for the RL agent, only for the PCA model. For example, there

was a significant decrease between the intermediate (group 2) and hard-to-detect (group 3) faults. Such behavior was not expected, but this analysis should be coupled with the analysis for FAR and MDR when inferring about the overall model performance. With this in mind, it is possible to conclude that, for all models, the level of difficulty had a significant impact on detection performance, as expected, for both PCA and RL models.

Table 10 – Average TTD values for the four fault groups. Values in parenthesis represent standard deviations (Gray cells mean FAR > 5% and bold values the best overall result given the fault group; "-" means undetected fault).

Group (difficulty)	RL (Arch1)	RL (Arch2)	PCA (T^2)	PCA (Q)
1 (Easy)	9.39 (29.21)	8.92 (11.49)	27.61 (29.64)	15.96 (15.46)
2 (Intermediate)	29.21 (20.82)	29.16 (21.43)	73.82 (32.81)	37.60 (22.15)
3 (Hard)	16.46 (13.53)	11.34 (13.75)	129.64 (111.62)	122.08 (112.05)
4 (Very-hard)	15.89 (8.91)	14.80 (8.91)	-	-

6.4. MODEL COMPLEXITY ANALYSIS

Regarding model complexity, two different Architectures, namely, Arch1 and Arch2, which differ in the number of parameters in the first hidden layer (256 neurons for Arch1 and 64 neurons for Arch2), were used to train different RL agents for the detection of twenty different faults. This particular choice had the objective of leveraging information to assess a possible relationship between model complexity and performance. **Figure 22** shows the performance of both neural network architectures on average FAR and MDR metrics. It can be seen that they showed similar behavior for both metrics, with Arch1 having better performance for some faults, but worse performance for others. Therefore, in this case, the greater complexity of Arch1 did not contribute to achieving better results. Other issues, mainly related to the structure of the data, can have a major influence on the complexity requirements. Moreover, since both Architectures produced similar results regardless of complexity, coupled with the fact that the Arch1-based model presented FAR values greater than 5% for one specific hard-to-detect fault (fault 19, with a false alarm rate of 8.95%), which did not happen for Arch2, the latter would be the architecture of choice for hypothetical applications.

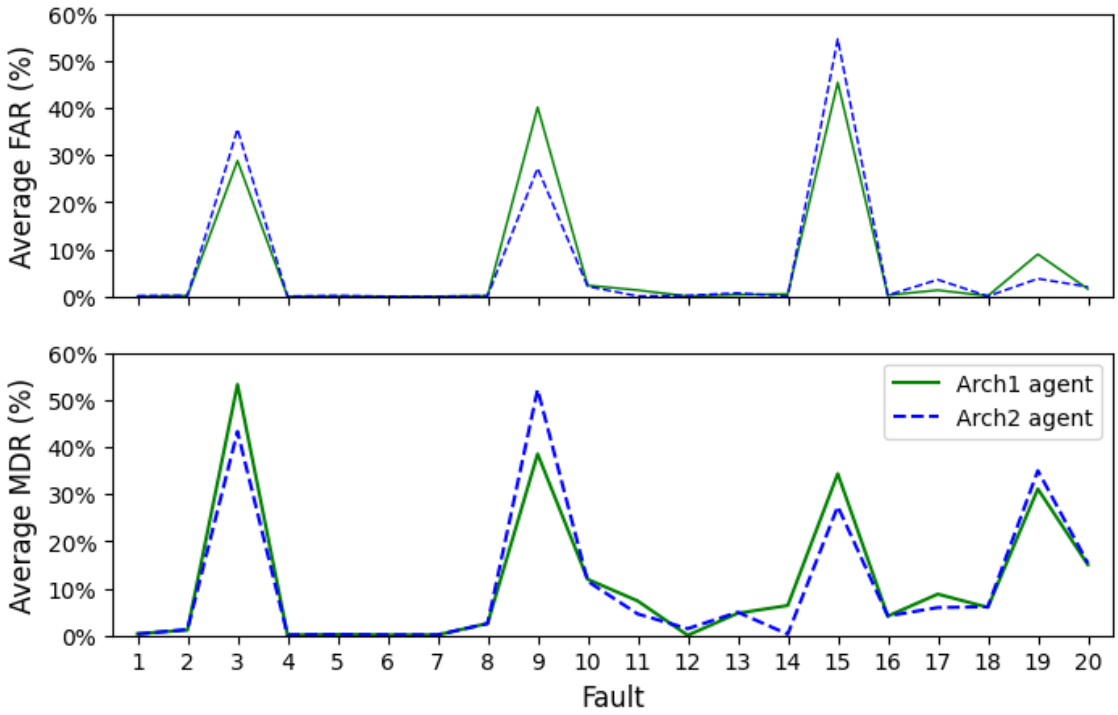


Figure 22: Average FAR (above) and MDR (below) values for Arch1 (continuous green line) and Arch2 (dashed blue line) RL agents across the twenty faults of the TEP benchmark.

7. CONCLUSIONS

The current work's objective originated from literature review, which indicated a rising number of papers dedicated to Deep Reinforcement Learning-based approaches to industrial fault detection in recent years, in an attempt to address the limitations of traditional Supervised Learning-approaches (BELHADI et al., 2021; DING et al., 2019; MALIK; SHARMA; MISHRA, 2020; QIAN; LIU, 2022; WANG; XUAN, 2021; WU et al., 2022, 2022; ZHONG; ZHANG; BAN, 2023). Moreover, the vast majority of these aforementioned applications were developed for industrial rotating machinery, which generate mostly univariate datasets, with simpler relationships among variables. Continuous chemical processes, on the other hand, have several unit operations and highly complex relationships among variables due, for example, to the non-linear characteristics of many parts of the process. Therefore, this project aimed to developed a tailored RL-based framework for fault detection for this type of industrial process. Usually, RL applications in continuous industrial chemical processes refer to process control. The framework was developed and validated for the Tennessee Eastman Process (TEP) benchmark, commonly explored by the process systems engineering community.

A total of forty RL agents, comprising two different neural network architectures and twenty different set of faults, were developed, trained, validated and tested. The results were then compared to a Principal Component Analysis (PCA) model,

often used in chemical process monitoring applications. For model evaluation and comparison, three different performance metrics were analyzed: false alarm rate (FAR), missed detection rate (MDR) and time-to-detection (TTD). The models were also analyzed from the perspective of detection difficulty, in which four different groups of faults were assembled in relation to their degree of detectability. The RL approach resulted in considerably better MDR results, especially regarding hard-to-detect faults, while keep FAR low. A similar behavior was encountered for TTD results. Through the proposed framework, faults that were initially difficult to detect were satisfactorily detectable.

REFERENCES

ABDI, H.; WILLIAMS, L. J. Principal component analysis. **WIREs Computational Statistics**, v. 2, n. 4, p. 433–459, 2010.

BAGNELL, J. A.; SCHNEIDER, J. G. **Autonomous helicopter control using reinforcement learning policy search methods**. Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164). **Anais**... Em: PROCEEDINGS 2001 ICRA. IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION (CAT. NO.01CH37164). maio 2001.

BELHADI, A. et al. Reinforcement learning multi-agent system for faults diagnosis of mircoservices in industrial settings. **Computer Communications**, v. 177, p. 213–219, 1 set. 2021.

BISHOP, C. M. Neural networks and their applications. **Review of Scientific Instruments**, v. 65, n. 6, p. 1803–1832, 1 jun. 1994.

BRAGA, A. DE P.; LUDERMIR, T. B.; CARVALHO, A. C. P. DE L. F. **Redes neurais artificiais: teoria e aplicações**. Rio de Janeiro: LTC, 2000.

BRO, R.; K. SMILDE, A. Principal component analysis. **Analytical Methods**, v. 6, n. 9, p. 2812–2831, 2014.

CHIANG, L. H.; RUSSELL, E. L.; BRAATZ, R. D. **Fault Detection and Diagnosis in Industrial Systems**. [s.l.] Springer Science & Business Media, 2000.

DING, Y. et al. Intelligent fault diagnosis for rotating machinery using deep Q-network based health state classification: A deep reinforcement learning approach. **Advanced Engineering Informatics**, v. 42, p. 100977, 1 out. 2019.

DOWNS, J. J.; VOGEL, E. F. A plant-wide industrial process control problem. **Computers & Chemical Engineering**, Industrial challenge problems in process control. v. 17, n. 3, p. 245–255, 1 mar. 1993.

GHESU, F. C. et al. **An Artificial Agent for Anatomical Landmark Detection in Medical Images**. (S. Ourselin et al., Eds.)Medical Image Computing and Computer-Assisted Intervention - MICCAI 2016. **Anais**...: Lecture Notes in Computer Science.Cham: Springer International Publishing, 2016.

GUTIERREZ-ROJAS, D. et al. Improving Fault Detection in Industrial Processes by Event-Driven Data Acquisition. **IEEE Access**, v. 10, p. 80918–80931, 2022.

HARRIS, C. S. et al. Big Data in Oncology Nursing Research: State of the Science. **Seminars in Oncology Nursing**, Special issue: Big Data in Cancer Care. v. 39, n. 3, p. 151428, 1 jun. 2023.

HOTELLING, H. Analysis of a complex of statistical variables into principal components. **Journal of Educational Psychology**, v. 24, p. 417–441, 1933.

HSU, F.-H. IBM's Deep Blue Chess grandmaster chips. **IEEE Micro**, v. 19, n. 2, p. 70–81, mar. 1999.

ISERMANN, R.; BALLÉ, P. Trends in the application of model-based fault detection and diagnosis of technical processes. **Control Engineering Practice**, v. 5, n. 5, p. 709–719, 1 maio 1997.

JIANG, Y. et al. Data-Driven Flotation Industrial Process Operational Optimal Control Based on Reinforcement Learning. **IEEE Transactions on Industrial Informatics**, v. 14, n. 5, p. 1974–1989, maio 2018.

JOY, M.; KAISARE, N. S. Approximate dynamic programming-based control of distributed parameter systems. **Asia-Pacific Journal of Chemical Engineering**, v. 6, n. 3, p. 452–459, 2011.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement Learning: A Survey. **Journal of Artificial Intelligence Research**, v. 4, p. 237–285, 1 maio 1996.

KESAVAN, P.; LEE, J. H. Diagnostic Tools for Multivariable Model-Based Control Systems. **Industrial & Engineering Chemistry Research**, v. 36, n. 7, p. 2725–2738, 1 jul. 1997.

LEE, J. M.; LEE, J. H. Approximate dynamic programming-based approaches for input–output data-driven control of nonlinear processes. **Automatica**, v. 41, n. 7, p. 1281–1288, 1 jul. 2005.

LYMAN, P. R.; GEORGAKIS, C. Plant-wide control of the Tennessee Eastman problem. **Computers & Chemical Engineering**, v. 19, n. 3, p. 321–331, 1 mar. 1995.

MAICAS, G. et al. **Deep Reinforcement Learning for Active Breast Lesion Detection from DCE-MRI**. (M. Descoteaux et al., Eds.)Medical Image Computing and Computer Assisted Intervention − MICCAI 2017. **Anais**...: Lecture Notes in Computer Science.Cham: Springer International Publishing, 2017.

MALIK, H.; SHARMA, R.; MISHRA, S. Fuzzy reinforcement learning based intelligent classifier for power transformer faults. **ISA Transactions**, v. 101, p. 390–398, 1 jun. 2020.

MARTINEZ-MARIN, T. **On-line optimal motion planning for nonholonomic mobile robots**. Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. **Anais**... Em: PROCEEDINGS 2006 IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2006. ICRA 2006. maio 2006.

MITCHELL, T. M. **Machine Learning**. 1ª edição ed. New York: McGraw-Hill Science/Engineering/Math, 1997.

MNIH, V. et al. Human-level control through deep reinforcement learning. **Nature**, v. 518, n. 7540, p. 529–533, fev. 2015.

MORDATCH, I. et al. **Combining model-based policy search with online model learning for control of physical humanoids**. 2016 IEEE International Conference on Robotics and Automation (ICRA). **Anais**... Em: 2016 IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION (ICRA). maio 2016.

MURPHY, K. P. **Machine Learning: A Probabilistic Perspective**. [s.l.] MIT Press, 2012.

NOORI, A.; SADRNIA, M. A.; SISTANI, M. BAGHER N. **Glucose level control using Temporal Difference methods**. 2017 Iranian Conference on Electrical Engineering (ICEE). **Anais**... Em: 2017 IRANIAN CONFERENCE ON ELECTRICAL ENGINEERING (ICEE). maio 2017.

PARK, Y.-J.; FAN, S.-K. S.; HSU, C.-Y. A Review on Fault Detection and Process Diagnostics in Industrial Processes. **Processes**, v. 8, n. 9, p. 1123, set. 2020.

PEDOTA, M. Big data and dynamic capabilities in the digital revolution: The hidden role of source variety. **Research Policy**, v. 52, n. 7, p. 104812, 1 set. 2023.

PRAVEEN KUMAR, D.; AMGOTH, T.; ANNAVARAPU, C. S. R. Machine learning algorithms for wireless sensor networks: A survey. **Information Fusion**, v. 49, p. 1–25, 1 set. 2019.

PUTERMAN, M. L. **Markov Decision Processes: Discrete Stochastic Dynamic Programming**. [s.l.] John Wiley & Sons, 2014.

QIAN, G.; LIU, J. Development of deep reinforcement learning-based fault diagnosis method for rotating machinery in nuclear power plants. **Progress in Nuclear Energy**, v. 152, p. 104401, 1 out. 2022.

ROJAS, R. The Backpropagation Algorithm. Em: ROJAS, R. (Ed.). **Neural Networks: A Systematic Introduction**. Berlin, Heidelberg: Springer, 1996. p. 149–182.

RUSSELL, E. L.; CHIANG, L. H.; BRAATZ, R. D. Fault detection in industrial processes using canonical variate analysis and dynamic principal component analysis. **Chemometrics and Intelligent Laboratory Systems**, v. 51, n. 1, p. 81–93, 8 maio 2000.

SILVER, D. et al. Mastering the game of Go without human knowledge. **Nature**, v. 550, n. 7676, p. 354–359, out. 2017.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning, second edition: An Introduction**. 2. ed. [s.l.] MIT Press, 2018.

TESAURO, G. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. **Neural Computation**, v. 6, n. 2, p. 215–219, 1 mar. 1994.

THIERY, C.; SCHERRER, B. Improvements on Learning Tetris with Cross Entropy. **ICGA Journal**, v. 32, n. 1, p. 23–33, 1 jan. 2009.

TSENG, H.-H. et al. Deep reinforcement learning for automated radiation adaptation in lung cancer. **Medical Physics**, v. 44, n. 12, p. 6690–6705, dez. 2017.

WANG, Z.; XUAN, J. Intelligent fault recognition framework by using deep reinforcement learning with one dimension convolution and improved actor-critic algorithm. **Advanced Engineering Informatics**, v. 49, p. 101315, 1 ago. 2021.

WATKINS, C. Learning From Delayed Rewards. 1 jan. 1989.

WATKINS, C. J. C. H.; DAYAN, P. Q-learning. **Machine Learning**, v. 8, n. 3, p. 279–292, 1 maio 1992.

WIERING, M.; VAN OTTERLO, M. (EDS.). **Reinforcement Learning: State-of-the-Art**. Berlin, Heidelberg: Springer, 2012. v. 12

WU, Z. et al. A deep reinforcement transfer convolutional neural network for rolling bearing fault diagnosis. **ISA Transactions**, v. 129, p. 505–524, 1 out. 2022.

XAVIER, G. M.; DE SEIXAS, J. M. **Fault Detection and Diagnosis in a Chemical Process using Long Short-Term Memory Recurrent Neural Network**. 2018 International Joint Conference on Neural Networks (IJCNN). **Anais**... Em: 2018 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IJCNN). jul. 2018.

ZHANG, A. et al. **Dive into Deep Learning**. [s.l: s.n.].

ZHAO, Y.; KOSOROK, M. R.; ZENG, D. Reinforcement learning design for cancer clinical trials. **Statistics in Medicine**, v. 28, n. 26, p. 3294–3315, 2009.

ZHONG, X.; ZHANG, L.; BAN, H. Deep reinforcement learning for class imbalance fault diagnosis of equipment in nuclear power plants. **Annals of Nuclear Energy**, v. 184, p. 109685, 1 maio 2023.

ZOU, J.; HAN, Y.; SO, S.-S. Overview of Artificial Neural Networks. Em: LIVINGSTONE, D. J. (Ed.). **Artificial Neural Networks: Methods and Applications**. Methods in Molecular Biology™. Totowa, NJ: Humana Press, 2009. p. 14–22.

APPENDIX – COMPLEMENTARY PLOTS

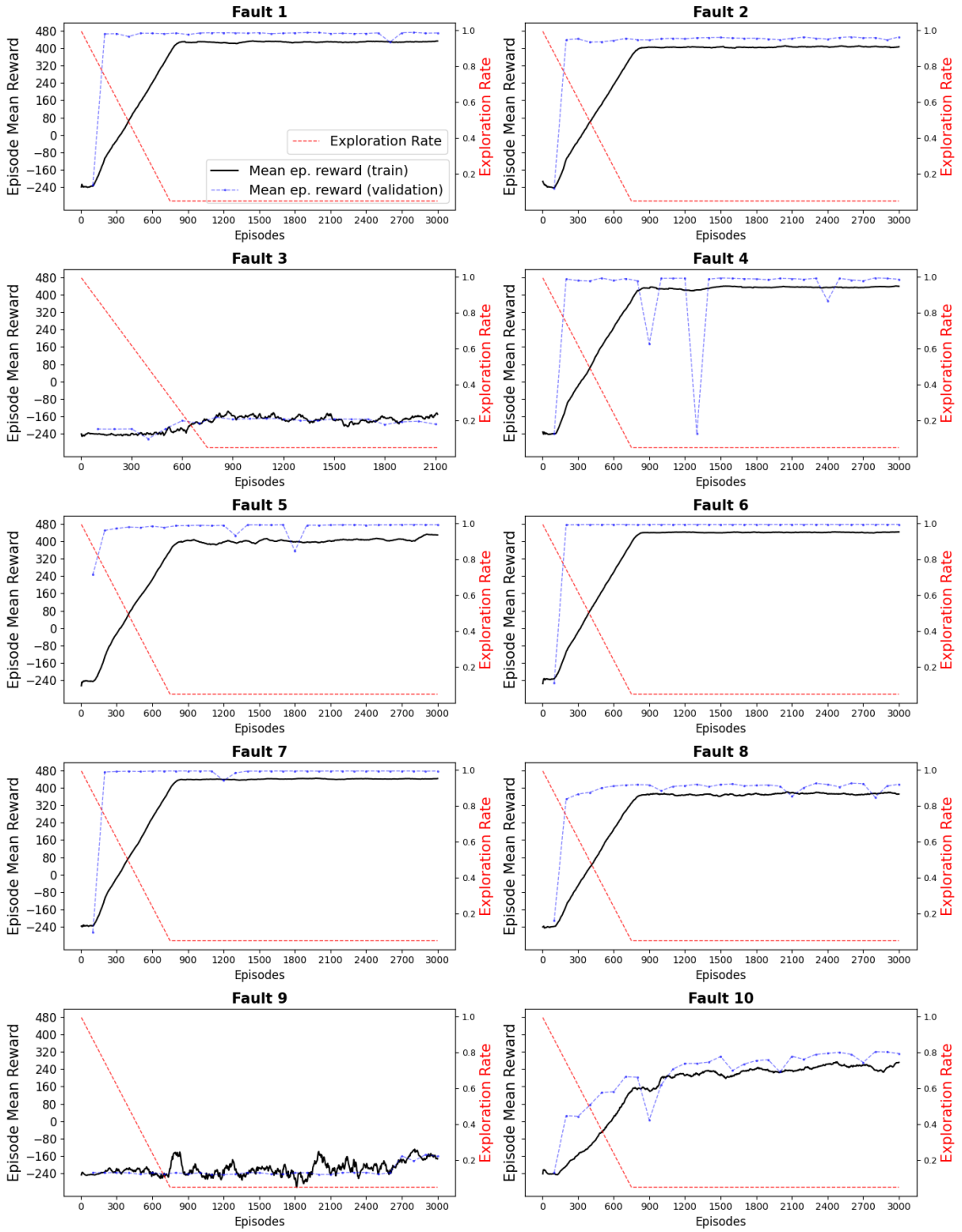


Figure 23: Learning curves curves (faults 1 to 10) for the RL-based model (Arch1).

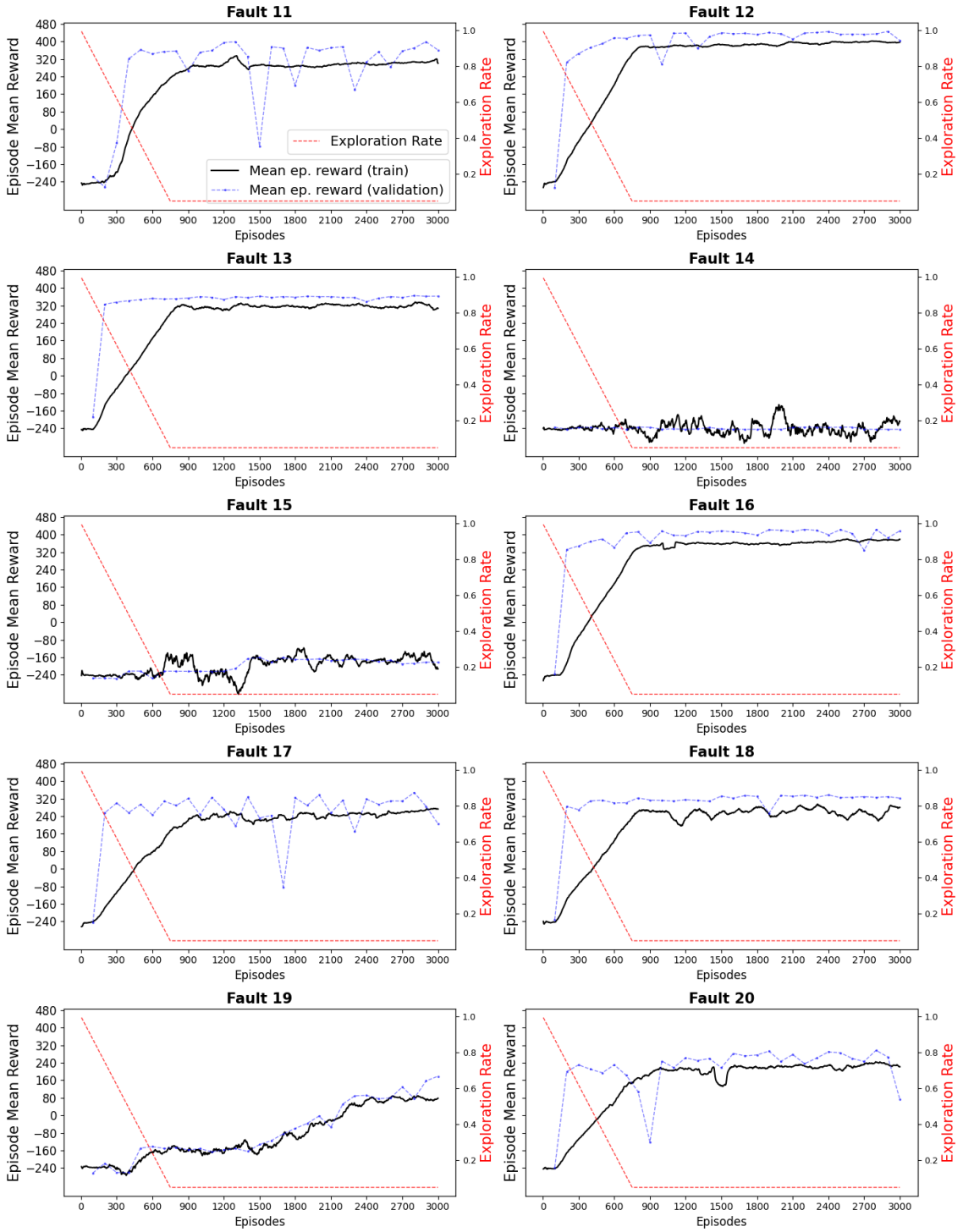


Figure 24: Learning curves curves (faults 11 to 20) for the RL-based model (Arch1).

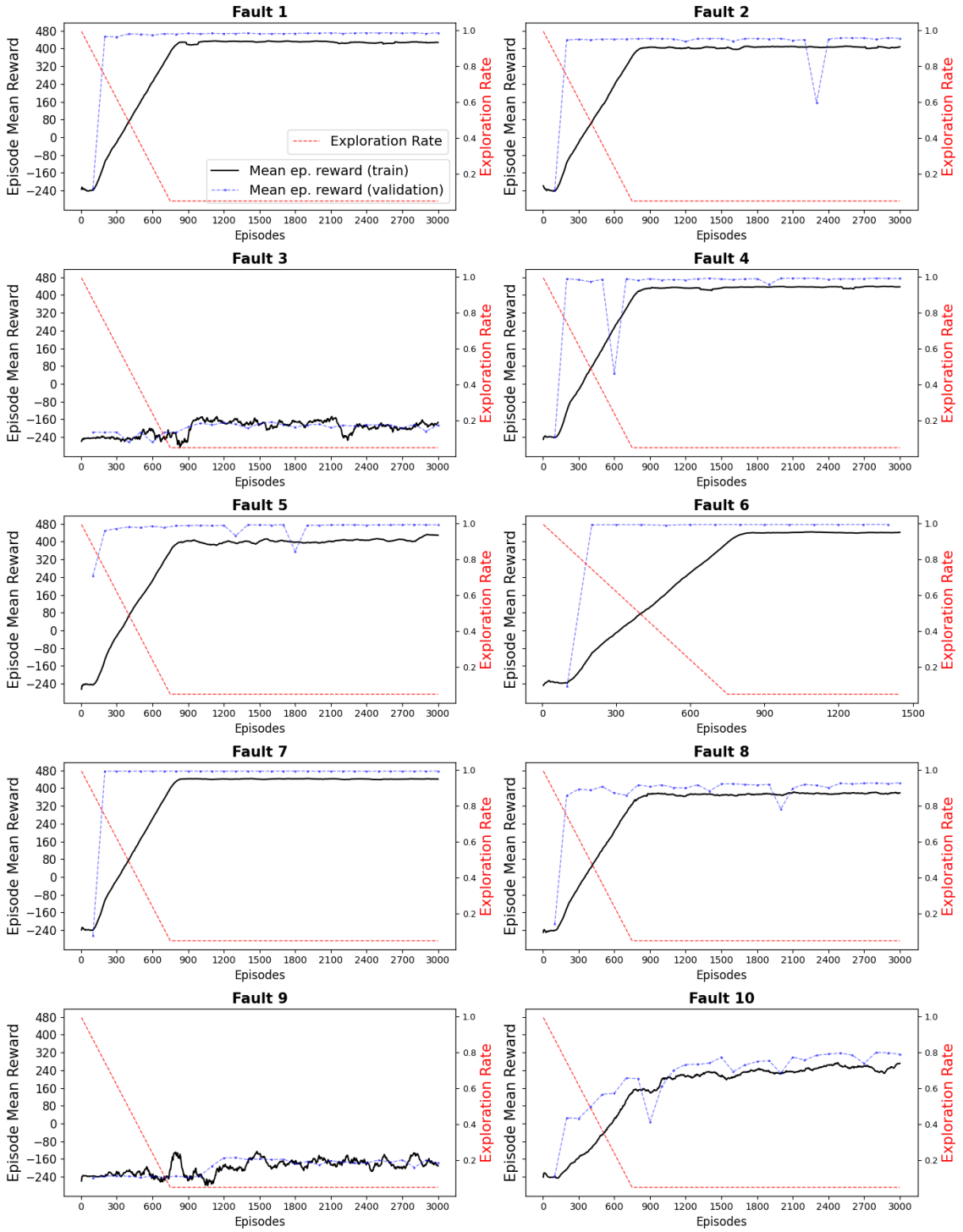


Figure 25: Learning curves curves (faults 1 to 10) for the RL-based model (Arch2).

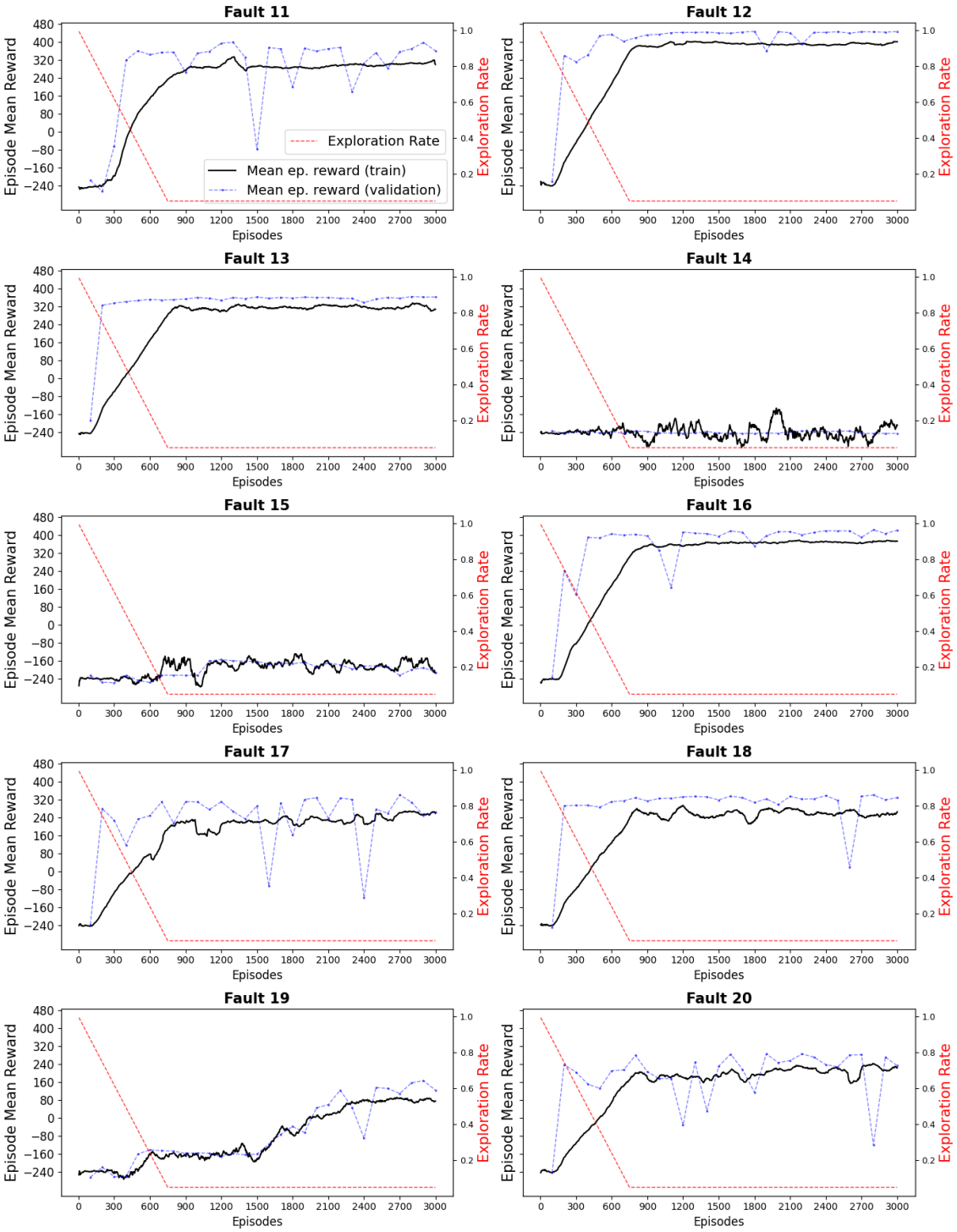


Figure 26: Learning curves curves (faults 11 to 20) for the RL-based model (Arch2).

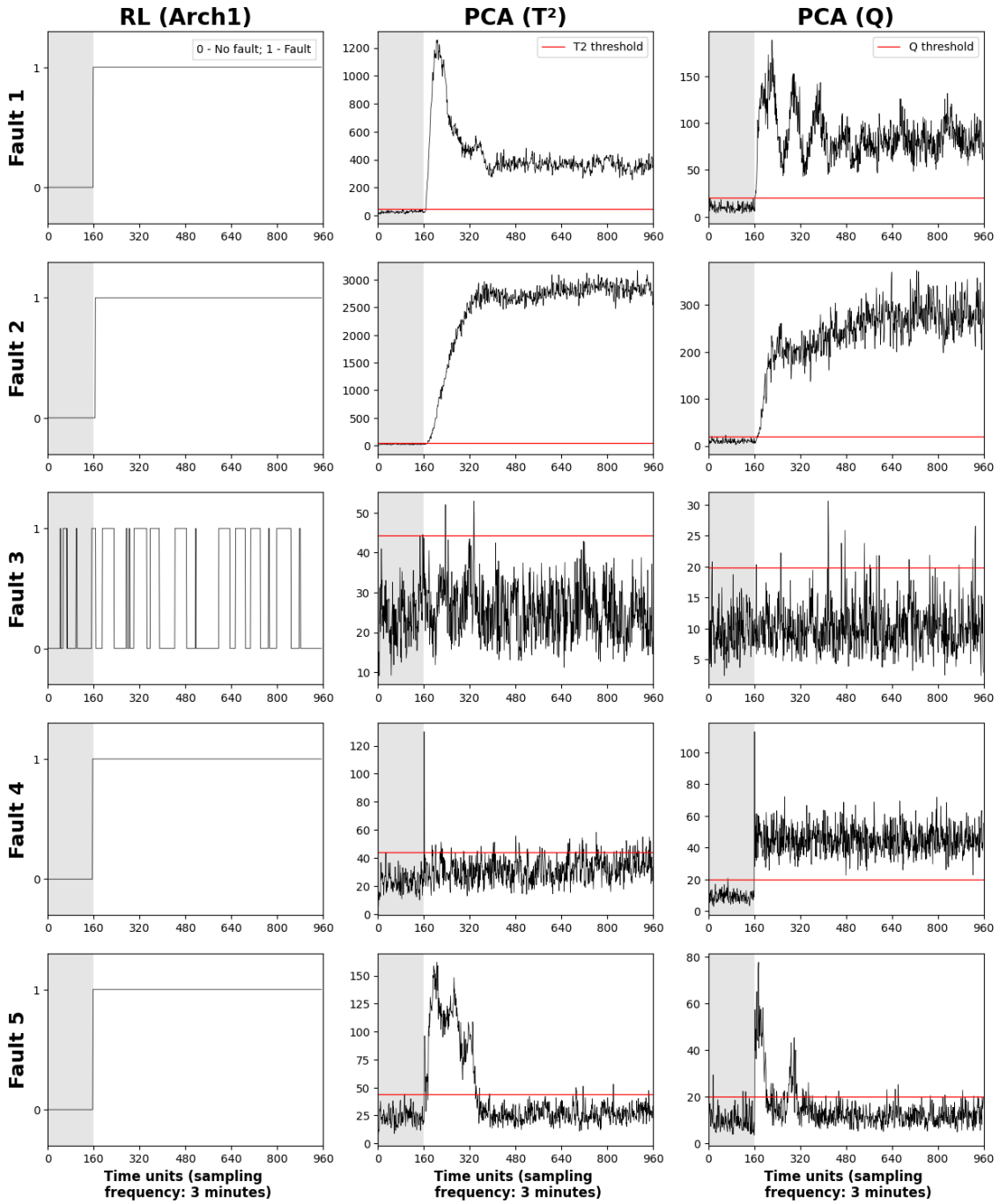


Figure 27: Detection plots (faults 1 to 5) for the RL-based model (Arch1), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.

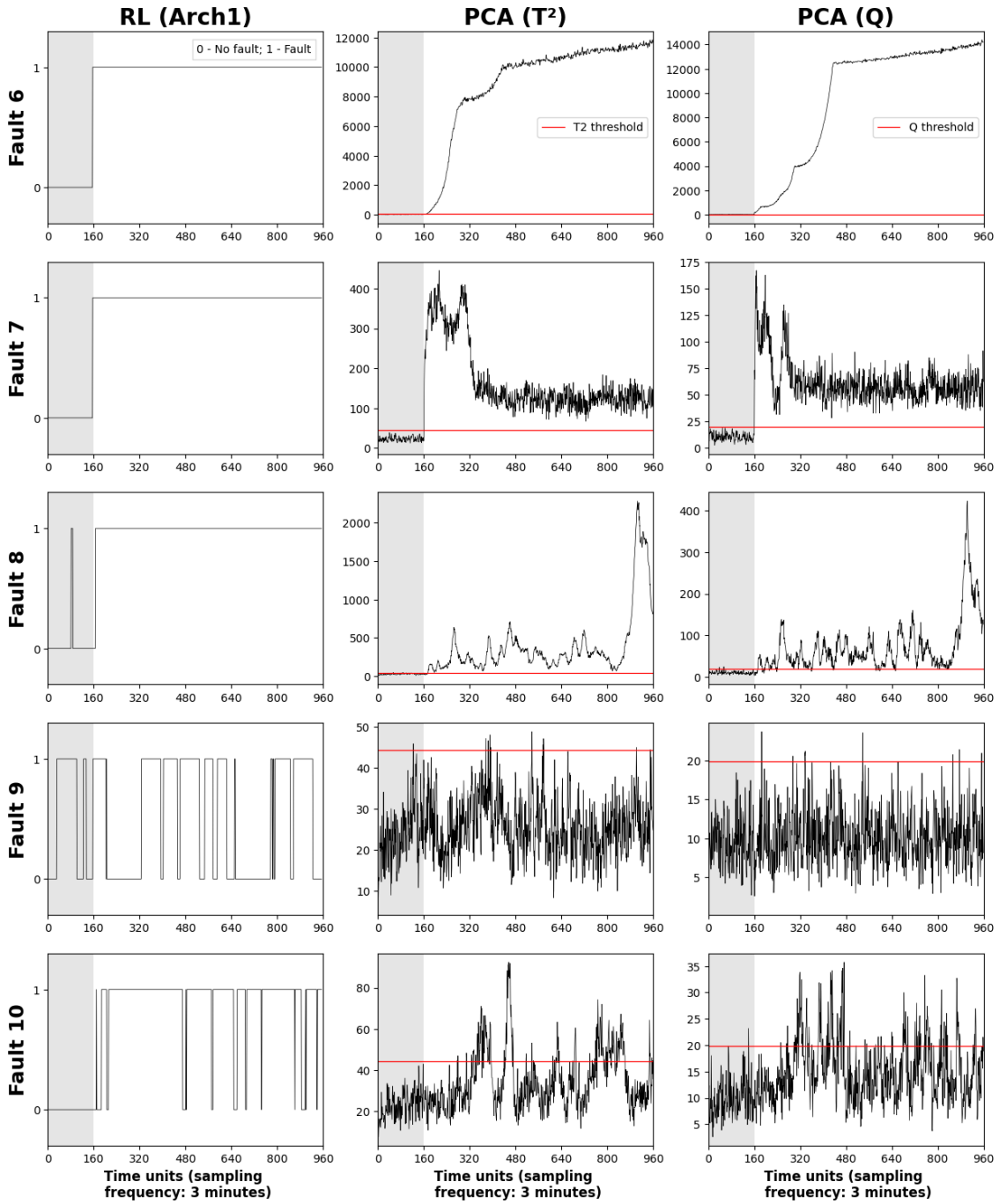


Figure 28: Detection plots (faults 6 to 10) for the RL-based model (Arch1), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.

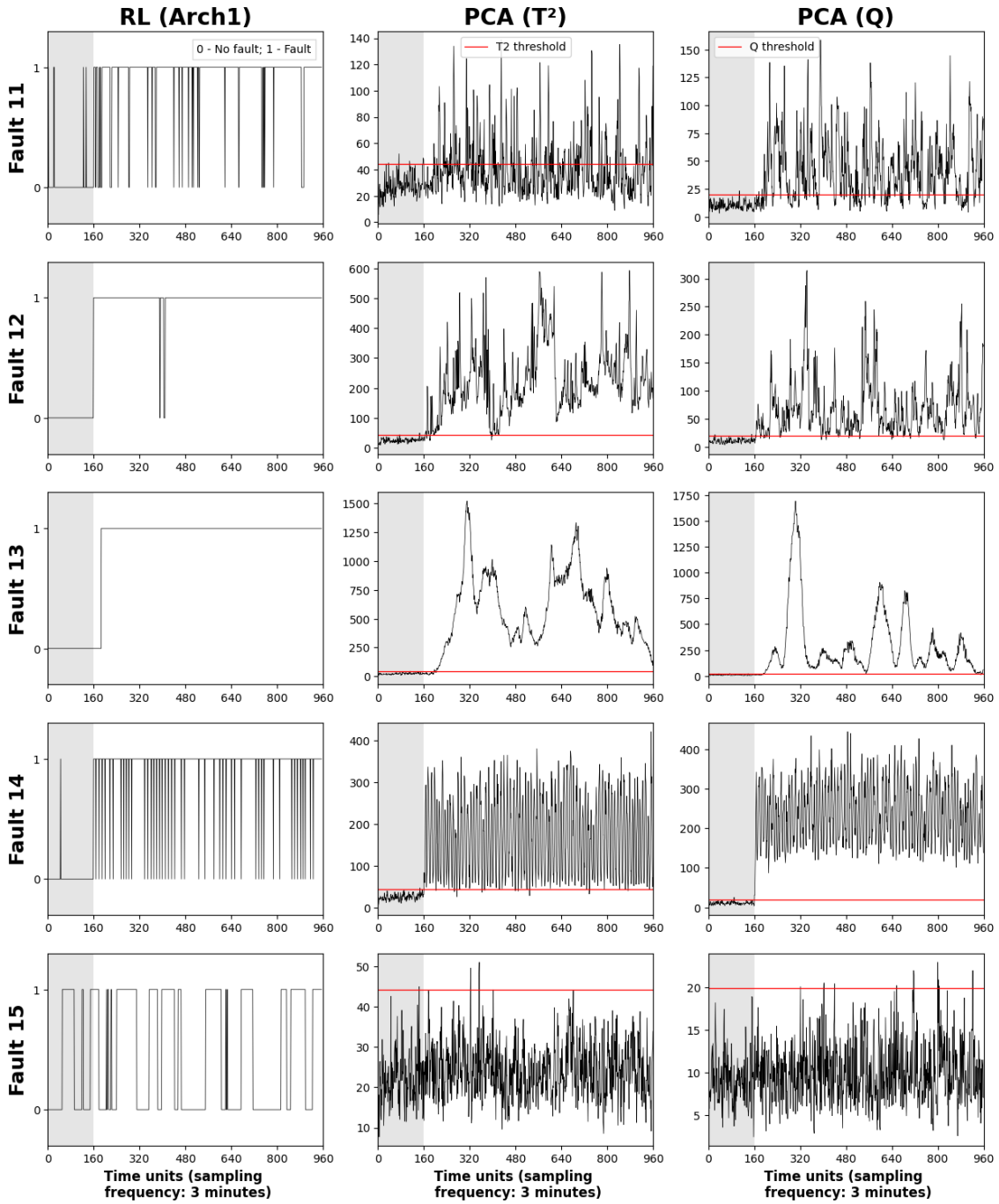


Figure 29: Detection plots (faults 11 to 15) for the RL-based model (Arch1), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.

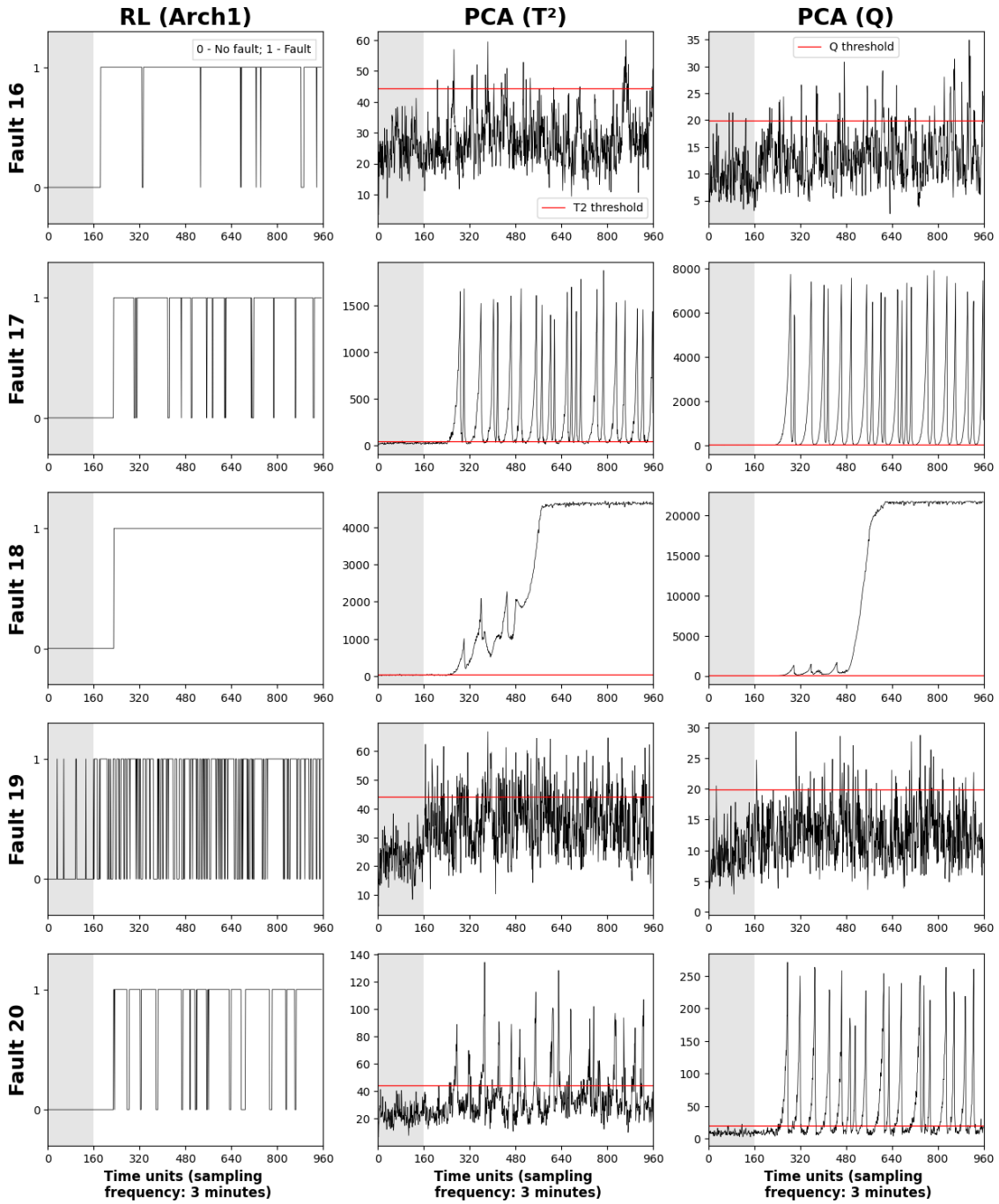


Figure 30: Detection plots (faults 6 to 10) for the RL-based model (Arch1), the T^2 test for the PCA baseline model, and the Q test for the PCA baseline model. The gray background indicates normal operation, whereas the white background indicates fault operation. The red horizontal lines in the PCA columns refer to the corresponding threshold value for the statistic.