# A Deep Reinforcement Learning Approach for Real-time Sensor-Driven Decision Making and Predictive Analytics

**Article** *in* Computers & Industrial Engineering · July 2020

DOI: 10.1016/j.cie.2020.106600

Computers & Industrial Engineering 147 (2020) 106600

Contents lists available at ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

# A deep reinforcement learning approach for real-time sensor-driven decision making and predictive analytics

Erotokritos Skordilis[a], Ramin Moghaddass[b,*]

[a] National Renewable Energy Laboratory, Golden, CO 80401, USA
[b] Department of Industrial Engineering, University of Miami, Coral Gables, FL 33146, USA

ARTICLE INFO

*Keywords:*
Particle filters
Deep reinforcement learning
Real-time control
Decision-making
Remaining useful life estimation

ABSTRACT

The increased complexity of sensor-intensive systems with expensive subsystems and costly repairs and failures calls for efficient real-time control and decision making policies. Deep reinforcement learning has demonstrated great potential in addressing highly complex and challenging control and decision making problems. Despite its potential to derive real-time policies using real-time data for dynamic systems, it has been rarely used for sensor-driven maintenance related problems. In this paper, we propose two novel decision making methods in which reinforcement learning and particle filtering are utilized for (i) deriving real-time maintenance policies and (ii) estimating remaining useful life for sensor-monitored degrading systems. The proposed framework introduces a new direction with many potential opportunities for system monitoring. To demonstrate the effectiveness of the proposed methods, numerical experiments are provided from a set of simulated data and a turbofan engine dataset provided by NASA.

## 1. Introduction

The development of advanced sensors, smart devices, and wireless/remote sensing technologies over the past decade has contributed significantly to the advancement of real-time monitoring and control and the development of smart asset management tools and techniques. Development of real-time control and decision making policies is challenging for complex systems that are monitored by many sensors where only imperfect information describing their latent dynamics can be observed over time. Recent advances in deep reinforcement learning (DRL), which is a combination of reinforcement learning (RL) and deep neural networks (DNNs), have shown its potential to be the new generation of decision-making frameworks for complex systems. DRL agents can typically learn by themselves to establish successful optimal policies for gaining maximum long-term rewards. Although applications that have employed DRL have shown very promising results, sensor-driven maintenance decision making based on DRL has been rarely studied and deserves more research to be fully applicable and beneficial for condition-monitored degrading systems.

The key factor in developing real-time control and decision making policies is the ability to represent system dynamics with mathematical equations. State-space models (SSM) are known as powerful classes of time-series models for the analysis of dynamical systems that have been widely applied in various application domains. Since latent degradation states are not directly observable, it is not possible to derive any policy that can directly map latent states to the action space. Instead, the decision policy can be defined over the belief space, which is the probability distribution of latent states. Bayesian modeling is one of the most powerful frameworks for analyzing SSMs and providing inference for latent states. Among all Bayesian filtering techniques, particle filtering (PF) has received a significant amount of attention over the past years because of its interesting properties that overcome some of the fundamental challenges of traditional models, such as Kalman filter. We will develop DRL frameworks that can utilize the outcomes of the particle filtering for latent state inference and transform them into the maintenance action space.

In this paper, we develop two new decision making frameworks that utilize both Bayesian filtering and DRL in an intelligent and coordinated manner. First, a decision making framework is developed that can use sensor data to determine when to perform maintenance prior to a system failure based on the relative relationship between the costs of replacement and failure. Second, a prognostics framework for remaining useful life estimation is developed that can generate warnings based on the relative relationship between early warning and late warning. The original inputs for both frameworks are multi-dimensional sensor data that are stochastically related to the system's latent degradation state. The frameworks developed in this paper do not depend on the structure of the system's dynamics and have no prior

* Corresponding author at: 279 McArthur Engineering Building, Department of Industrial Engineering, University of Miami, Coral Gables, FL 33146, USA.
*E-mail address:* ramin@miami.edu (R. Moghaddass).

https://doi.org/10.1016/j.cie.2020.106600
Received 5 September 2019; Received in revised form 2 May 2020; Accepted 6 June 2020
Available online 04 July 2020

distributional assumptions. This makes these frameworks very generic and thus more applicable for a wide range of degrading systems. It is important to note that using RL frameworks with Bayesian filtering has important advantages compared to RL frameworks without Bayesian filtering that use sensor data as inputs. First, using Bayesian filtering allows an explicit quantification of uncertainties unlike standard DRL that uses direct sensor observations as point values without incorporating uncertainty into the sensor data. Second, sensor observations often depend only on the current status of the system and do not contain information on the history of the degradation process. However, Bayesian filters can infer the current state of the system based on the history of sensor measurements and the latent degradation process. Third, Bayesian filtering can combine complex multi-dimensional sensor data and thus using its output as the input for training a reinforcement learning framework is computationally more appealing. This is because the dimension of the output of Bayesian filtering equals the dimension of the corresponding discretized latent variables, which can be significantly smaller than the original dimension of sensor data. In addition to the above characteristics, there are other general advantages of using Bayesian reasoning in RL, such as dealing with action-selection (exploration/exploitation) as a function of the uncertainty and incorporating prior knowledge and regularization (Ghavamzadeh, Mannor, Pineau, & Tamar, 2015).

The main contributions of this paper can be summarized as follows: **First**, we propose a real-time control and decision making framework for system maintenance that combines the capabilities of Bayesian filtering and DRL. **Second**, we introduce an original Bayesian filtering-based prognostics framework that employs DRL for remaining useful life (RUL) estimation and generating various types of warnings depending on the user's preferences. **Third**, we demonstrate that DRL can infer stochastic system control policies using a stochastic representation of the system's latent states over time without having to develop full sweeps of the state space or exhaustive back-ups. To the best of our knowledge, this is the first time that such frameworks are developed for degrading systems. The DRL frameworks developed in this paper can generalize historical sensor observations to new and previously unseen states and system conditions to make real-time actions. Also, the frameworks can be used for large problems with large state spaces, including continuous space systems. The RL frameworks are model-free and no explicit distribution for system dynamics, state transition, sensor data, and risk/reward functions are required. This helps develop new diagnostics and prognostics tools without having to define too many unrealistic and hard-to-justify distributional and parametric assumptions normally made for mathematical convenience.

This article is structured as follows: In Section 2, we provide a review of the current literature on condition-based maintenance, Bayesian filtering, and work on reinforcement learning. The mathematical foundations of the hybrid state-space models, neural networks, and DRL are presented in Section 3. Then, the structure of the proposed decision making frameworks is presented in Section 4. Numerical experiments using simulated and real-case degradation data are presented in Section 5. Finally, our conclusions and directions for future work are discussed in Section 6.

## 2. Related work

### *2.1. System monitoring with sensor data*

Timely and accurate maintenance plans for deteriorating systems are known as one of the most important decision making topics in complex systems, particularly in critical systems with costly repair and maintenance. Condition-based maintenance (CBM) is known as one of the most commonly used maintenance approaches for deteriorating systems under sensor observation (Ko & Byon, 2017). In Bousdekis, Magoutas, Apostolou, and Mentzas (2018), a comprehensive review of various methods dealing with prognostic-based decision support for CBM within complex systems is provided. These methods have generally diversified characteristics and their implementation varies depending on the application. A review on data acquisition, health indicator generation, health stage division, and RUL prediction in machinery prognostics can be found in Lei et al. (2018).

### *2.2. Decision making and remaining useful life prediction*

A full CBM policy involves two types of decision making and analytical problems: when to terminate the operation for repair/replacement and, at any given point of time, how many more cycles are left until the system fails. The second question can be framed as a decision making problem as follows: For any threshold *d*, when should a warning be issued so that d cycles are left until the failure point? Recent literature on CBM has revealed various approaches that try to address these questions. In Zhao, Gaudoin, Doyen, and Xie (2019), the authors proposed an inspection/replacement policy that depends on both degradation data and non-periodic inspections during system operation that can have either positive or negative effects. The study in Sun, Ye, and Chen (2017) presented an inspection/replacement CBM strategy for a multi-unit system, with the objective of deriving optimal decisions for maintenance cost minimization. In addition to models that can help find the optimal replacement/repair time, a majority of the literature on system prognostics is devoted to RUL prediction. In Tsoutsanis and Meskin (2017), the problem of RUL prediction was discussed for an industrial gas turbine under transient conditions. In reference (Cheng, Qu, & Qiao, 2018), a new fault prognostic and RUL prediction method for wind turbine gearbox using adaptive neuro-fuzzy inference and particle filtering was proposed. In reference (Kontar et al., 2017), the problem of inaccurate RUL predictions for early failed systems are discussed by proposing a signal-based RUL prediction method focusing on imbalanced historical data. A novel approach for combining different categories of observations for inducing a proper health index and RUL prediction was proposed in Baraldi, Bonfanti, and Zio (2018). In Fang, Gebraeel, and Paynabar (2017), a scalable semi-parametric statistical framework that utilizes high dimensional monitoring signals for real-time RUL prediction of partially degraded systems was proposed.

### *2.3. Models to represent system dynamics*

Successful implementation of any CBM policy requires appropriate mathematical models that can represent the latent degradation of a system with condition monitoring data and can provide a reasonable representation of system dynamics. State-space modeling (SSM) is known as a powerful class of time-series models for the analysis of dynamical systems and has been widely applied in various application domains. There is a growing body of literature that recognizes the importance of SSMs due to their mathematical convenience and reasonable way to model and analyze dynamic systems with multiple inputs and outputs. The most popular method to analyze SSMs is recursive Bayesian filtering, such as Kalman filters (Wang & Tsui, 2018) and particle filters (PF) (Song, Liu, Yang, & Peng, 2017). Most of these studies take into consideration only the latent degradation process and a single level for the observation process. Inspired by the concept of hybrid systems (Orchard & Vachtsevanos, 2009), a discrete mode variable can be defined to represent discrete levels of hidden working conditions or faults, (e.g., normal and faulty) at which the dynamics of the system may change. Such a variable can also be responsible for incorporating switching behaviors of system's dynamics within a system's life cycle. Hybrid state-space models (HSSMs) have also been introduced in the areas of prognostics and RUL estimation (e.g., see Orchard & Vachtsevanos, 2009). HSSMs have general properties and are broadly applicable to many areas, such as target tracking, robot movement, and sensor fault detection and identification (Park, Pil Hwang, Kim, & Kang, 2010; Wei, Huang, & Chen, 2009).

### 2.4. Decision making and analytical techniques

Many condition-based maintenance decision policies have been developed using dynamic programming and Markov decision processes (MDPs). MDPs are characterized by actions and rewards that define the state of the system at any given time. In Olde Keizer, Teunter, and Veldman (2016), an optimal CBM policy for multi-unit systems was developed while taking into account both redundancy through a *k*-out-of-*N* structure and economic dependencies. The study in Byon and Ding (2010) presented an optimal policy based on backward dynamic programming for choosing cost-effective maintenance actions for wind turbine gearbox operations. In Zhou, Xi, and Lee (2009), an opportunistic preventive maintenance scheduling method for multi-unit series systems was proposed based on DP. Despite its strong mathematical structure, DP is limited since it requires full access to the entire state transition and reward models necessary to solve the Hamilton-Jacobi-Bellman equation. Because these models are not always known, it is necessary to employ methods capable of obtaining optimal control policies using model-free approaches. For that reason, RL can be used to obtain control policies without relying on any parametric model form. Instead, approximation functions, such as neural networks, can be used. In RL, a software agent takes actions within an environment in order to maximize a cumulative reward.

Despite its power, RL has received very limited attention in the area of real-time control of deteriorating systems. The study in Xanthopoulos, Kiatipis, Koulouriotis, and Stieger (2017) utilized RL for obtaining optimal maintenance and control policies for deteriorating, stochastic production, and inventory systems. In Wang, Wang, and Qi (2016), the authors developed a semi-MDP model for describing the degradation of a flow line system and applied an RL algorithm to obtain a control-limit maintenance policy for each machine in the flow line. Reference (Dohi & Okamura, 2016) considered an operational software system with multi-stage degradation levels due to software aging. An RL model was developed to derive an optimal dynamic software rejuvenation policy by maximizing steady-state system availability using a semi-Markov decision process. These studies, while demonstrating the potential of RL in CBM decision policies, do not make use of function approximators but rather use tables in which value function solutions are stored. While such an approach is simple, it cannot be used when the state space is very large. Also, it cannot be used for states that have not been previously observed.

Reinforcement learning has been employed for decision making and control in various fields. In Allen, Roychowdhury, and Liu (2018), the authors considered a preventive maintenance problem for university cybersecurity. Because of the limited amount of data obtained from intrusion detection systems, they utilized Monte-Carlo RL due to its ability to generalize MDPs for addressing parametric uncertainty using appropriate prior probability distributions. In Shiue, Lee, and Su (2018), a RL framework was proposed for a real-time control process aiming at improving the production performance of manufacturing machines in a small factory. Sahebjamnia, Tavakkoli-Moghaddam, and Ghorbani (2016) proposed an approach for real-time control of the quality of a continuous chemical production line using a fuzzy Q-learning multi-agent quality control system. In Ko (2019), a mathematical model was proposed for optimal decision-making regarding wireless charging of electric tram systems. They also used RL for controlling the proposed solution from the genetic algorithm chromosome. The model managed to reach a solution comparable to the one derived from applying a mathematical optimization solver. None of the above-mentioned RL methods used any kind of neural networks as the Q-function approximator. In all of these methods, a mathematical model was employed as a value function (model-based) for an infinite time horizon. In our work, we use model-free (neural networks) RL for the action-value function in a finite time horizon. Also, a Q-table was used in the above references to store possible action-state pairs that denote agent moves in a fully observable environment.

Advancements in computing capabilities have increased the popularity of deep learning (Goodfellow, Bengio, & Courville, 2016) that can employ neural network architectures with two or more hidden layers or specialized designs that allow recurrence and convolution. The advancements in deep neural networks have led to the development of DRL, with significant breakthroughs such as the deep Q-network (Mnih et al., 2015) and AlphaGo (Silver et al., 2016). DRL can develop control policies for previously intractable problems in the domains of video gaming and robot movement. Recent reviews on the various applications of DRL can be found in Arulkumaran, Deisenroth, Brundage, and Bharath (2017). Over the last few years, more studies have been conducted on using DRL for maintenance decision making. In Andriotis and Papakonstantinou (2019), a DRL framework referred to as the deep centralized multi-agent actor-critic that provides life-cycle maintenance/replacement and inspection policies for multi-component systems was proposed. The algorithm was developed as a multi-agent DRL, in which each agent developed a policy for each of the subsystems. Our work is different from Andriotis and Papakonstantinou (2019) since we focus on a single unit system with a latent continuous degradation state. Reference (Liu, Chen, & Jiang, 2020) utilized an actor-critic DRL for dynamically optimizing a selective maintenance strategy for a multi-state system with binary-capacitated components that maximizes the expected number of successful future missions of multi-component systems. The binary states of all components can be completely determined at the beginning of each mission. Our work is different from Liu et al. (2020) since we focus on a single unit system with a latent continuous degradation state, which is only partially observable through sensor data. The structure of our work is different from both (Liu et al., 2020 & Andriotis & Papakonstantinou, 2019) because we utilized a pure off-policy method (deep Q-networks) to develop the optimal policies while both papers have utilized actor-critic networks, which are hybrid and employ policy gradients that operate in the policy space to provide the optimal actions. In Wei, Bao, and Li (2020), a DRL framework for the automated policy making of bridge maintenance actions was introduced and then an optimization model for learning the optimal maintenance policy was proposed. They considered a non-Bayesian approach with known states in which discrete structural rates are known by inspections. None of the above studies considered DRL for estimating the remaining useful life of the systems and developing a warning policy for maintenance. Compared to (Andriotis & Papakonstantinou, 2019; Liu et al., 2020; Wei et al., 2020), our model considered more generic system's dynamics, that is, a hybrid state-space model with multiple stochastic layers (e.g., degradation process, operating condition, and hazard process).

### 2.5. Summary of the literature gap and contributions of the paper

We believe that there is great potential in the recent advancements in artificial intelligence that has not been fully realized in the domain of sensor-driven prognostics and system maintenance decision-making. Previous studies in control and maintenance decision-making applications have assumed the availability of fully observable systems and/or full knowledge of the state transition and risk/reward models, both of which are unrealistic in complex systems with dynamic structures and various sources of uncertainty. For that reason, it is necessary to develop real-time control approaches that take into consideration the uncertainty of system dynamics and are tractable without making too many distributional and parametric assumptions. Also, new methods are needed to find the optimal maintenance time and predict remaining useful life that depend mainly on sensor observations and can be sufficiently trained with historical data. In this paper, we introduce new frameworks to develop real-time control and decision making policies using (a) Bayesian filtering to make inference regarding the latent state of the system and (b) DRL in which agent actions are chosen based on a stochastic environment with states inferred by Bayesian filtering. This paper develops new sensor-driven RL-based decision making frameworks in which the environment that the agent observes is entirely stochastic and inferred using Bayesian filtering and sensor observations. We will demonstrate that this framework can be used to generate warning signals based on the number of cycles left to failure and also predict remaining life anytime during the system's operation.

## 3. System characteristics and monitoring tools

Before introducing the proposed frameworks, a brief review of the system under study and assumptions made in the paper as well as the structure of the state-space modeling, Bayesian filtering, and neural network architectures are provided in this section.

### *3.1. State-space modeling to represent system dynamics*

State-space modeling of dynamic systems offers a flexible and generic mathematical framework that can be used to describe the dynamics of partially observed degrading systems, where the degradation process is hidden and can only be inferred indirectly from sensor observations. Almost all state-space models used for system health monitoring follow a generic stochastic filtering structure in a dynamic state-space form of

$$x_t = f_x(x_{t-1}, \boldsymbol{u}_t, \theta_x, \epsilon), \tag{1}$$

$$\boldsymbol{y}_t = f_y(x_t, \boldsymbol{u}_t, \theta_y, \delta), \tag{2}$$

where $x_t$ denotes the one-dimensional continuous-value hidden degradation process and $\boldsymbol{y}_t$ denotes a multivariate ($m$-dimensional) observation process at time *t*. Stochastic functions *f:* display the evolution of model variables and $\boldsymbol{\Theta} = \{\theta_x, \theta_y\}$ are model parameters that characterize the behavior of these functions. Variable $\boldsymbol{u}_t$ represents a $d$-dimensional vector of controllable and uncontrollable operational inputs that are problem specific. Both processes are perturbed by statistical noise denoted by $\epsilon$ and $\delta$ for the system and observation noises, respectively. These noise processes are assumed to be time-invariant and follow normal distributions with zero means and covariance matrices $Q$ and $\boldsymbol{R}$ as follows:

$$\epsilon \sim \mathcal{N}(0, Q), \quad \delta \sim \mathcal{N}(\boldsymbol{0}, \mathbf{R}).$$

Despite their capabilities, generic SSMs cannot capture multimodal hidden dynamics that also consider discrete-valued processes. For that reason, hybrid state-space models (HSSM) for health monitoring are developed. Discrete hidden states usually denote the operating condition of the system at any time, which can be either binary (normal, faulty) or can take multiple discrete values, whereas the continuous hidden state still represents the overall system health (i.e., degradation) status. A generic HSSM with multiple stochastic layers, including a latent continuous degradation state process $x_t$, a latent discrete state process representing the operating condition of the system $c_t$ (i.e., faulty or normal), a latent hazard process $\lambda_t$ representing the system's probability of failure, an observation process $y_t$ representing sensor data, and the working status process $o_t$ representing the overall system's working conditions, is considered in this paper. Following the same principle presented in Eqs. (1)–(2), the structure of the HSSM used in this paper is as follows:

$$c_t = f_c(c_{t-1}, \boldsymbol{u}_t, \theta_c), \tag{3}$$

$$x_t = f_x(x_{t-1}, c_t, \boldsymbol{u}_t, \theta_x), \tag{4}$$

$$\boldsymbol{\lambda}_t = f_\lambda(x_t, c_t, \boldsymbol{u}_t, \theta_\lambda), \tag{5}$$

$$\boldsymbol{y}_t = f_y(x_t, c_t, \boldsymbol{u}_t, \theta_y), \tag{6}$$

$$o_t = f_o(\boldsymbol{\lambda}_t, \theta_o). \tag{7}$$

For simplicity, noise terms are removed from the above equations. To represent the nonlinear relationship between sensor observations and latent states as described in Eq. (6), an Extreme Learning Machine (ELM) single-layer feed-forward neural network that was first introduced in Huang, Zhu, and Siew (2006) is used in this paper. ELM does not need any distributional dependencies between variables and can be used to model high-dimensional and even dependent and correlated sensor measurements. Using this approach, we can overcome the obstacle of finding an appropriate parametric function with some distributional assumptions to represent the observation process.

We use parameter vector $\theta_y$ to represent the ELM neuron weights. The frameworks proposed in this paper are very general and can be used for any forms of functions *f*:. One may simply modify the causal relationships between model variable by changing the inputs and outputs in functions *f*. However, to better explain these frameworks, we consider the following structure for system dynamics. For the degradation process, we adopted a unit-less, one-dimensional health index that degrades exponentially. We consider two operating conditions for the latent discrete state process $c_t$, namely, normal operation and faulty operation. Finally, the latent hazard process $\lambda_t$ taking values between 0 and 1 is used to represent the conditional probability of failure in the time interval $(t-1, t)$, given that the system survived up until time $t-1$, and variables $x_t$, $c_t$, and $\boldsymbol{u}_t$ are known. We utilized *logistic regression* for characterizing function $f_\lambda$ due to its mathematical flexibility to formulate the probability of failure under the binary assumption of being either at a normal or a failed state given the covariates. The structure of the hazard process can be shown as follows:

$$\lambda_t = [1 + \exp[-(ax_t + \beta c_t + \boldsymbol{\gamma}\boldsymbol{u}_t + \beta_0)]]^{-1},$$

where $a, \beta, \boldsymbol{\gamma}$ denote the covariate coefficients that together with $\beta_0$ create the parameter set $\theta_\lambda$. Hazard process $\lambda_t$ and the working status observation process $o_t$ (Eq. (7)) are conditionally dependent, that is, the probability of a system failing ($o_t = 1$) is $\lambda_t$ as

$$p(o_t = 1 | o_{1:t-1} = \boldsymbol{0}) = \lambda_t.$$

Since no other parameters relate $\lambda_t$ and $o_t$, the parameter vector $\theta_o$ is empty.

Fig. 1 presents an overview of the HSSM structure used in this paper that enables monitoring the dynamic behavior of systems over time taking into account multiple dynamic measurements collected from available sources (e.g., sensors) while the system is operating. Hidden states are given in circles and the observable measurements are given in rectangles. Operating inputs are given as diamonds. The hidden/latent layers $c_t$ and $x_t$ are not observable over time. These latent layers can change the behaviour of sensor outputs, the hazard process, and the overall working status of the system. The structure used in the paper to represent the system dynamic is very general and can cover many available structures used in the literature. Many condition-monitored systems with latent degradation states and sensor observations can be modeled with this generic framework. Once the framework is trained

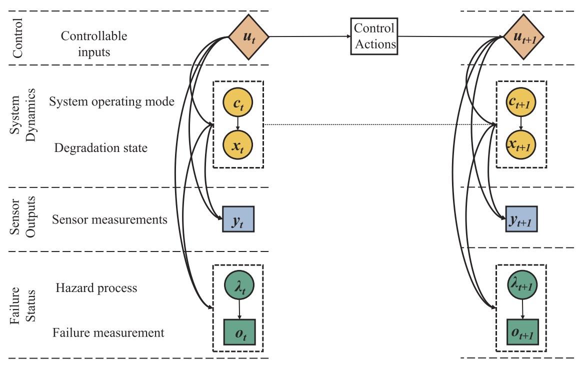

**Fig. 1.** Structure of the proposed SSM. Circles represent latent states, rectangles represent observations, and diamonds represent operating inputs. The causal relationships (vertical edges) between model variables in a generative form, as well as their temporal dependencies (horizontal edges), are shown by arrows.

with past data, one can use real-time sensor observations to infer the latent degradation status of the system.

### 3.2. Bayesian filtering to monitor latent states

Since the latent states for the HSSM discussed earlier are not observable over time, we need to be able to compute the posterior distribution of the latent variables, which is equivalent to the probability distribution $p(x_t, c_t, \lambda_t|\mathbf{y}_{1:t}, o_{1:t})$ at any time $t$. Using the two steps of prediction and update, we can calculate this measure recursively. In the prediction step, the prior distribution $p(x_t, c_t, \lambda_t|\mathbf{y}_{1:t-1}, o_{1:t-1})$ is computed based on the filtering distribution $p(x_{t-1}, c_{t-1}, \lambda_{t-1}|\mathbf{y}_{1:t-1}, o_{1:t-1})$ as follows:

$$p(x_t, c_t, \lambda_t|\mathbf{y}_{1:t-1}, o_{1:t-1}) = \iiint p(c_t|c_{t-1})p(x_t|x_{t-1}, c_t)p(\lambda_t|x_t, c_t) p(x_{t-1}, c_{t-1}, \lambda_{t-1}|\mathbf{y}_{1:t-1}, o_{1:t-1})dx_{t-1}dc_{t-1}d\lambda_{t-1}.$$

In the update step, a new measurement vector $\{\mathbf{y}_t, o_t\}$ is collected and then using the Baye's rule, the aforementioned prior distribution updates the posterior as follows:

$$p(x_t, c_t, \lambda_t|\mathbf{y}_{1:t}, o_{1:t}) \propto p(\mathbf{y}_t|x_t, c_t)p(o_t|\lambda_t)p(x_t, c_t, \lambda_t|\mathbf{y}_{1:t-1}, o_{1:t-1}). \quad (8)$$

For notational convenience, we refer to the set of latent states as $\mathbf{z}_t = \{x_t, c_t, \lambda_t\}$ and the set of observable measurements as $Y_t = \{\mathbf{y}_t, o_t\}$. Particle filtering is the most commonly used Bayesian filtering method for state inference in nonlinear state-space models and can also take hybrid forms when multiple layers of latent state processes are considered. Particle filtering represents a sequential Monte Carlo method, which performs inference in state-space models, and is used to estimate the hidden state at time $t$ given all measurements $\mathbf{y}_{1:t}$ up to time $t$. The idea behind particle filtering is to approximate the posterior distribution at time $t$, $p(\mathbf{z}_t|Y_{1:t})$, with a weighted set of samples $\{x_{1:t}^i, c_{1:t}^i, \lambda_{1:t}^i, w_t^i\}_{i=1}^{N_s}$, also called particles. The notation $N_s$ denotes the number of particles. The most commonly used particle filtering algorithm is the sequential importance sampling (SIS) in which the target posterior distribution $p(\mathbf{z}_t|Y_{1:t})$ is approximated using samples drawn from a proposal distribution $q(\mathbf{z}_t|Y_{1:t})$ from which it is easier to sample compared to the target distribution. To accommodate the variation between the two distributions, every sample $\mathbf{z}_t^i = \{x_t^i, c_t^i, \lambda_t^i\}$ has to be weighted by:

$$w_t^i \propto w_{t-1}^i \frac{p(\mathbf{y}_t|x_t^i, c_t^i)p(o_t|\lambda_t^i)p(x_t^i|x_{t-1}^i, c_t^i)p(c_t^i|c_{t-1}^i)p(\lambda_t^i|x_t^i)}{q(x_t^i|x_{1:t-1}^i, c_{1:t}^i, \mathbf{y}_{1:t})q(c_t^i|c_{1:t-1}^i, \mathbf{y}_{1:t})q(\lambda_t^i|x_{1:t}^i, o_{1:t})}.$$

After weights are obtained, they are normalized to ensure the sum is 1. By applying importance sampling to the target posterior distribution, we get an approximation as follows:

$$\hat{p}\left(\mathbf{z}_{1:t}\middle|Y_{1:t}\right) \approx \sum_{i=1}^{N_s} w_t^i \delta_{x_{1:t}, c_{1:t}, \lambda_{1:t}}\left(x_{1:t}^i, c_{1:t}^i, \lambda_{1:t}^i\right), \quad (9)$$

where $\delta_{\cdot}$ denotes the delta function centered at particles $\mathbf{z}_{1:t}^i = \{x_{1:t}^i, c_{1:t}^i, \lambda_{1:t}^i\}$ and $N_s$ denotes the number of samples drawn. It has been proved theoretically that as $N_s \to \infty$, the target distribution approximates the actual proposal distribution (Chen, 2003). Algorithm 4 in Appendix A provides a summary of the hybrid particle filtering to be used to estimate latent states $\mathbf{z}_{1:t}^i$ $(1 \leqslant i \leqslant N_s)$ and the uncertainties associated with such estimates. In summary, we can conclude that when using a set of known inputs and observable measurements from available sensors, Bayesian filtering can be used to estimate the probability distribution of latent states over time based on the structure given in Section 3.1. Such a stochastic distribution of latent states will be later employed as an input for decision making.

### 3.3. Deep reinforcement learning

Based on the framework discussed earlier in this section, latent states are not directly observable, and thus it is not possible to derive any policy or decision making framework that can directly map latent states to the action space. Instead, the decision policy can be defined over the belief space, that is, the probability distribution of latent states. The belief state at any time point can be computed from the vector of observation history using the approach discussed in Section 3.2. To derive a decision policy as the transformation from the belief space to the action space, reinforcement learning will be used as a sequential decision-making approach that follows a discrete time stochastic control process. Reinforcement learning can develop an autonomous *agent* that learns a system's behavior depending on the surrounding *environment* using a single feedback indicator known as a *reward*. The ultimate objective of RL is to perform a certain set of *actions* that will lead to reward maximization. RL is often trained through a trial-and-error process beginning from a random initial setting. RL algorithms are generally modeled by utilizing *Markov decision processes* (MDPs) since the obtained observation from the environment satisfies the *Markov property*, that is, the future state only depends on the current state/action pair. The corresponding MDP can be described by a 5-tuple $(\mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{Z} = \{\boldsymbol{\zeta}_t\}$ is the set of belief states, $\mathcal{A} = \{\alpha^1, ..., \alpha^Z\}$ is the set of $Z$ different control actions, $\mathcal{P}$ represents the probability transition equation $p(\boldsymbol{\zeta}_{t+1}|\boldsymbol{\zeta}_t, \alpha_t)$, $\mathcal{R}(\boldsymbol{\zeta}_t, \alpha_t)$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor. At time $t$, the agent receives the state of the environment (i.e., belief state $\boldsymbol{\zeta}_t$) and chooses a control action $\alpha_t$. The system reacts to the action and then moves to the next state $\boldsymbol{\zeta}_{t+1}$, where it receives a reward $r_{t+1}$. Both $\boldsymbol{\zeta}_{t+1}$ and $r_{t+1}$ are provided to the agent as feedback from the environment. At each training iteration, the agent updates either the *value function* $\mathcal{V}(\boldsymbol{\zeta})$ or the *action-value function* $Q(\boldsymbol{\zeta}, \alpha)$ based on a specific *policy* that maps states $\boldsymbol{\zeta} \in \mathcal{Z}$ to actions $\alpha \in \mathcal{A}$ (Kaelbling, Littman, & Moore, 1996). When an action-value function is considered, the expected outcome of being at state $\boldsymbol{\zeta}$ and taking action $\alpha$ following policy $\pi$ is the cumulative discounted reward for all future action-state pairs. The optimal policy $\pi^*$ can be achieved by greedily implementing actions that maximize the action-value function as

$$Q^{\pi}(\boldsymbol{\zeta}, \alpha) = \mathbb{E}_{\pi}(r_t + \gamma Q^{\pi}(\boldsymbol{\zeta}_{t+1}, \alpha_{t+1})|\boldsymbol{\zeta}_t = \boldsymbol{\zeta}, \alpha_t = \alpha). \quad (10)$$

Based on this method, the decision rule for any state $\boldsymbol{\zeta}$ is to choose an action that maximizes $Q^{\pi^*}(\boldsymbol{\zeta}, \alpha)$. In the standard Q-learning implementation, Q-values are stored in a table and then the best policy of actions over the set of states can be determined. One cell is required per combination of state-action. This implementation is not amenable to continuous state and action problems. As the implementation of the finite horizon solution of the described Q-learning with continuous state-space is computationally intractable, it is proposed to discretize the belief space, so that the state-space becomes finite and discrete based on the discretization step, which can be cross-validated by data. Since developing implicit Q-tables is not an option, generalization with Q-function approximation is required to obtain the optimal policy. Neural and deep neural networks (DNNs) have been proven as efficient function approximators leading to the domain of DRL, also called *deep Q-learning* (Mnih et al., 2015). Based on DRL, we utilize a neural network as an approximator for the action-value function as

$$\widehat{Q}(\boldsymbol{\zeta}, \alpha; \theta) \approx Q(\boldsymbol{\zeta}, \alpha),$$

where $\theta$ denotes the neuron weights in the corresponding neural networks. This approximation, known as *Q-network* (Mnih et al., 2013), improves the action-state function through minimizing a series of loss functions

$$\mathcal{L}(\theta_k) = \mathbb{E}_{\boldsymbol{\zeta}, \alpha}[(\mathcal{Y}_k - \widehat{Q}(\boldsymbol{\zeta}, \alpha; \theta_k))^2], \quad (11)$$

where $\mathcal{Y}_k = \mathbb{E}_{\boldsymbol{\zeta}'}[r_k + \gamma \max_{\alpha'} \widehat{Q}(\boldsymbol{\zeta}', \alpha'; \theta_{k-1})|\boldsymbol{\zeta}, \alpha]$ is the network target of

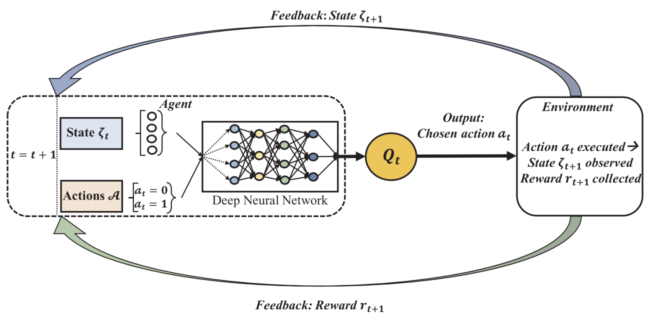

**Fig. 2.** DRL process. At time point $t$, the agent takes as input the $\{\zeta_t, \alpha\}$ pair ($\forall\ \alpha \in \mathcal{A}$) and returns the action with the highest reward. After an action is executed, the environment transits to a new state $\zeta_{t+1}$ and reward $r_{t+1}$ is collected. Both $\zeta_{t+1}$ and $r_{t+1}$ are provided as feedback to the agent.

iteration $k$. Minimization of Eq. (11) can be achieved through the gradient descent as shown in Eq. (12):

$$\nabla_{\theta_k}\mathcal{L}(\theta_k) = \mathbb{E}_{\zeta,\alpha}\left[\left(r_k + \gamma\max_{\alpha'}\widehat{Q}\left(\zeta', \alpha'; \theta_{k-1}\right) - \widehat{Q}\left(\zeta, \alpha; \theta_k\right)\right)\nabla_{\theta_k}Q\left(\zeta, \alpha; \theta_k\right)\right]. \tag{12}$$

In the case of DRL, it is computationally more effective to optimize loss functions by the stochastic gradient descent (Mnih et al., 2013). For notational convenience, we will use $Q$ instead of $\widehat{Q}$ to refer to the action-value function. Both state $\zeta$ and action $\alpha$ constitute the inputs of the neural network that estimate the action-value function $Q$. Fig. 2 demonstrates an outline of the DRL process with two possible actions. In such a process, an artificial agent interacts with its environment by gathering observations and taking actions sequentially. Once the action is executed, the state transitions to a new state and the agent obtains a reward. Bayesian filtering can combine complex multi-dimensional sensor data and thus use its output as the input for training a reinforcement learning framework. At the beginning of a decision epoch, the set of generated particles is utilized to estimate the probability distribution of these latent states (belief state).We will show in Section 4.1 the details of how the outcome of Bayesian filtering can be used as the input for the $Q$-network.

### 3.4. Neural network architectures in DRL

For $Q$-function approximation, we use three different neural network architectures for the action-value function approximator. The detailed structure of these networks is discussed in Appendix B. The first category is the Single-Hidden Layer Feed-Forward Neural network consisting of a collection of fully connected units called *artificial neurons* distributed in three sequential, densely connected layers, namely, input, hidden, and output layers. Neurons in each layer collect input signals and produce outputs by utilizing a nonlinear *activation function* over the sum of all inputs. The second category is the fully connected deep neural network (DNN), which almost follows the same principles as regular neural networks except that DNNs have at least two hidden layers. The third category is the deep recurrent neural network (DRNN), which is a deep neural network specifically designed for processing sequential data. In DRNN, connections between units form directed sequential graphs that allow them to display a temporal dynamic behavior.

### 3.5. Summary of Bayesian filtering and reinforcement learning

The dynamic of the system under study is assumed to follows a generic hybrid multi-layer state-space structure as discussed in Section 3.1. For such a system, the degradation states are only partially observable through sensor data. Bayesian Filtering (Section 3.2) can be used over time to process sensor observations sequentially to estimate the probability distribution of latent states (also called belief states). Such a distribution will be later used as an input to train a DRL process for decision making. The DRL process can use neural network structures as an approximator for learning the Q-values. The trained DRL agent can be employed at any decision making point to determine the best action only based on the belief state and according to the trained action-value function. The structure of the DRL and the details of how Bayesian filtering can generate input for DRL will be discussed in detail in Section 4.

## 4. Bayesian filtering-based DRL for maintenance decision making

In this section, we present two real-time control and decision making frameworks using DRL for optimal maintenance decisions for systems that are monitored with sensors. In Section 4.1, we illustrate how Bayesian filtering can be used to transform sensor data to the inputs required by DRL and how to train a DRL agent for a specific task. In Section 4.2, we describe a real-time system control framework in which an agent is trained to provide the best time to replace a faulty system so that the average maintenance cost is minimized. Then, in Section 4.3, we present a new decision making framework for RUL estimation based on a framework that utilizes particle filtering and DRL while taking into account the relative importance of under and over estimations.

### 4.1. From Bayesian filtering to DRL

As discussed earlier, the latent states are not directly observable for the systems under study and thus it is not possible to derive any policy that can directly map latent states to the action space. The multivariate sensor observations collected during system operation and the binary system working condition are the only observable inputs for any decision making task. The decision policy can be defined over the belief space, that is, the probability distribution of latent states. The belief state at any time point can be computed from the vector of observation history using particle filtering. The outcome of the particle filtering in Eq. (8) is a set of weighted particles that approximates a posterior belief about the latent system dynamics. The set of all particles at the beginning of any decision epoch can estimate the probability distribution of the latent states. Each particle vector represents a point estimate of the latent state $\boldsymbol{z}_t$. Since all particles at time $t$ can represent the probability distribution of the hidden state $\boldsymbol{z}_t$, given observations $Y_{1:t}$ at that time, we can create a discrete probability distribution. First, we divide the particle domain into $B$ intervals with discrete bounds $b_0, b_1, \ldots, b_B$. Intervals are assumed to be mutually exclusive and of equal length, covering the entire set of values that the particles can take. The values of $b_0$ and $b_B$ may be selected as the lowest and highest possible values of particles. Then we define the $d$th element of $\zeta_t$ as follows:

$$\zeta_t^d = \frac{\sum_{i=1}^{N_s}\mathbf{1}_{\{z_t^i \in (b_d, b_{d-1}]\}}}{N_s}, \quad 1 \leqslant d \leqslant B. \tag{13}$$

Based on the above transformation at every time point $t$, we obtain a discrete particle distribution with values $\zeta_t^1, \ldots, \zeta_t^B$ to be used as the system's *belief state*, that is, $\boldsymbol{\zeta}_t = [\zeta_t^1, \ldots, \zeta_t^B]$. Let us emphasize here that this quantization occurs separately for the particles of every hidden process in the state-space model. In other words, for the HSSM presented in Section 3.1, we consider discrete distributions for all three latent processes. The number of bins should be cross-validated according to the computational complexity and the ability to represent the particle distributions. Multiple bins should be defined for the continuous degradation process and hazard process while only two bins for the operating condition state process are needed. Fig. 3 provides a simple example of how particles can generate the empirical probability distributions to be later used as inputs for the $Q$-function approximator

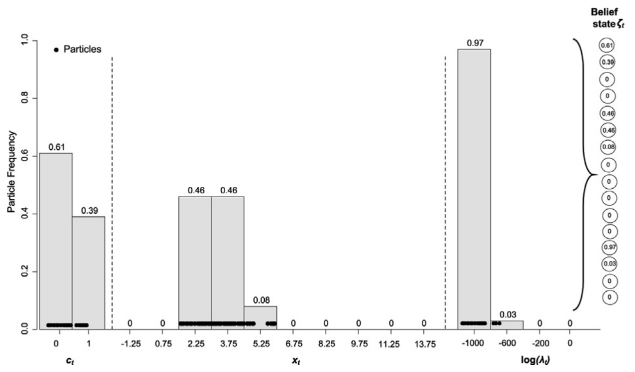

**Fig. 3.** Discretization of particle distribution. The vdots represent the actual particle values. The bars present the empirical probability of the corresponding range. These stochastic values are then used as part of the inputs needed for neural networks (e.g., belief state).

neural networks. For that reason, we first analyze the multivariate observation process through the hybrid particle filtering to estimate the posterior beliefs of the latent system dynamics. These estimates are then discretized to $B$ bins (as discussed above), providing the belief states of the latent system dynamics. As a result, we obtain a training dataset $\mathcal{I}$ of size $\boldsymbol{T}\times B$, with $\boldsymbol{T} = \sum_{j=1}^{N} T_j, j \in \left\{1, 2, \ldots, N\right\}$ denoting the sum of all lifetimes in $\boldsymbol{N}$. Dataset $\mathcal{I}$ contains the cardinality of all the environment states from all $\boldsymbol{N}$ systems that are used as inputs to train the DRL agent in the proposed framework. Each state is also associated with a set of possible control actions $\mathcal{A}$, which also acts as training inputs. The agent compares the action-value function $Q$ for every combination of state and possible actions and chooses the one that gives the best cumulative future reward for that state. Algorithm 1 provides the steps on how to train the DRL agent using available historical data recorded in the training set.

The first episode (iteration) starts at $t = 1$. Note that the agent action-value function output $Q$ is evaluated for all combinations of states and $Z$ actions in $\mathcal{A} = \alpha^1, \ldots, \alpha^Z$ as inputs for the Q-function approximator. The process repeats for all $t$. After the final $Q$-value at time $t = \boldsymbol{T}$ is obtained, the first DRL training iteration concludes. The process repeats for a number of iterations, or *episodes*, until no significant improvement in the agent's behavior is observed (i.e., $Q$-values converge). The number of episodes is defined by the user and can reach several hundred depending on the complexity of the problem. During the training process, the agent either selects the action randomly or based on the maximum value of those actions using the most updated information available to make a decision from the Q-function. The exploration step is critical as it ensures the agent explores states that otherwise may not be selected during the exploitation process. The exploitation/exploration learning rate $\epsilon$ determines the trade-off for these two types of selection. The training process can be stopped after a certain number of episodes or when a pre-defined criterion, such as not improving the loss function beyond a lower limit, is reached. We have utilized the two techniques of target network and experience replay (Mnih et al., 2013) that help with training efficiency, stability, and robustness. To facilitate agent training in Q-learning, a target network, which is a copy of the actual policy network (Q-function), can be used to generate the target Q-values necessary to calculate the loss for every action during training. With that, the Q-values remain constant for a number of episodes, leading to stable neural network training. Experience replay can also be used to store previously visited states in a replay buffer. Then, at every episode, a pre-specified number of states is sampled from the buffer to train the policy network. The sampling is necessary to break dependencies between states, thus making the training more stable. Both experience replay and target network are utilized in the paper to make the agent training process more efficient, stable, and robust. For the experience replay, we sample the entire training dataset due to the relatively small size of the state space and the training set. For the target network, we update all weights with the ones of the policy network on every episode.

**Algorithm 1.** Bayesian Filtering-based Deep Reinforcement Learning

```
Input: Full Sensor history of $N$ systems in the training set with age
    $T_j, \forall j \in \{1, 2, \ldots, N\}$.
for $j = 1$ to $N$ do
    Apply hybrid PF (Algorithm 4) to obtain particle values for the latent variables.
    Output $x_{1:T_j}^{1:N_S}, c_{1:T_j}^{1:N_S}, \lambda_{1:T_j}^{1:N_S}$.
end for
- Create the training dataset for DRL, which includes $\boldsymbol{T} \times \boldsymbol{B}$ data points, where
    $\boldsymbol{T} = \sum_{j=1}^{N} T_j, j \in \left\{1, 2, \ldots, N\right\}$.
- Introduce the set of control actions $\mathcal{A} = \{\alpha^1, \ldots, \alpha^Z\}$.
- Create the belief states $\zeta_{1:\boldsymbol{T}}$.
- Initialize random target values $Q_{1:\boldsymbol{T}}$.
- Set discount factor $\gamma$, initial neuron weights $\theta$, the number of episodes $E$, experience
    replay $\mathcal{G}$, and exploitation/exploration learning rate $\epsilon$. Fig. 4 provides a high-
    level overview of the training process.
for $e = 1$ to $E$ do
  for $j = 1$ to $N$ do
    for $t' = 1$ to $T_j$ do
      - Collect the belief state $\zeta_{t'}$.
      - With probability $\epsilon$ select a random action $\alpha_{t'} \in \mathcal{A}$
        (Exploration),
        otherwise select $\alpha_{t'} = argmaxA_{\alpha}Q(\zeta_{t'}, \alpha;\theta)$
        (Exploitation).
      - Execute action $\alpha_{t'}$, observe reward $r_{t'}$, and belief state $\zeta_{t'+1}$.
      - Store transition $(\zeta_{t'}, \alpha_{t'}, r_{t'}, \zeta_{t'+1})$ in $\mathcal{G}$.
      - Sample batch of transitions from $\mathcal{G}$.
      - Set $\mathcal{Y}_{t'} = \begin{cases} r_{t'}, & \text{if } t' = T_j. \\ r_{t'} + \gamma\max_{\alpha'}Q(\zeta_{t'+1}, \alpha';\theta), & \text{otherwise.} \end{cases}$
    end for
  end for
    - Perform stochastic gradient descent step (Eq. (12)) for neural network training.
    - New neuron weights $\theta'$ are obtained. Set $\theta = \theta'$.
end for
Output: The trained neural network and $Q$-functions.
```

### 4.2. Real-time control and decision making

In this section, we describe how the DRL structure can be used for decision making. Given a series of $N$ life trajectories (i.e., failed systems) in the training set in which each trajectory $j \in N$ consists of a sequence of continuous observations $\boldsymbol{y}_1^j, \ldots, \boldsymbol{y}_{T_j}^j$, discrete working status observations $o_1^j, \ldots, o_{T_j}^j$, and lifetime $T_j$, our objective is to develop a policy for real-time system control by training a DRL agent that minimizes the average maintenance cost. One of the most critical parts of the DRL algorithm is the shape of the reward function that provides rewards based on the current state of the system and the action taken. For the real-time control framework, we define a reward function that returns higher instant rewards when maintenance control action occurs before the system failure. Also, lower rewards are considered when a maintenance control action is chosen while the system is at its relatively early operating stages, when a total failure is unlikely. For instant reward allocation, we assume that the replacement of the device costs $c_r$ regardless of its status at the time of replacement. We also assume an additional cost for failure replacement $c_f$ that includes charges, such as extra labor, repair of damage to the rest of the system, and cost of downtime. Hence, for system $j$ with lifetime $T_j$, the instant reward at time $t \leqslant T_j$ is defined through negative penalty terms depending on the action chosen as follows:

$$r_t = \begin{cases} 0, & \alpha_t = \text{Do Nothing} \ \& \ \text{t} < T_j, \\ -\frac{c_r}{t}, & \alpha_t = \text{Replace} \ \& \ \text{t} < T_j, \\ -\frac{c_r + c_f}{T_j}, & \alpha_t = \text{Do Nothing} \ \& \ \text{t} = T_j, \\ -\frac{c_r + c_f}{T_j}, & \alpha_t = \text{Replace} \ \& \ \text{t} = T_j. \end{cases} \tag{14}$$

We consider two different actions, namely, "Do Nothing", which results in the system's operation continuing and "Replacement", which results in the system's operation being halted to perform the replacement. Both actions can take place either during system operations or after a system failure, in which case they both return a very low reward since we want to avoid system failure. If action "Do Nothing" takes place and the system is still operational ($t < T_j$), then no reward is returned. In this case, the agent is encouraged to let the system continue its operation. When action "Replace" is chosen by the agent while $t < T_j$, the reward depends on $t$. In this situation, the highest reward is given when $t = T_j - 1$. After the system fails, regardless of the chosen action, the system is subject to the negative reward of $c_r + c_f$ divided by the age of the system. Algorithm 2 summarizes the structure of the framework that shows how a trained DRL agent can transform a set of sensor data into an optimal action: whether to terminate the operation and replace. Once the Q-function is fully trained at any time point $t$, the following decision rule can be used at time point $t$:

$$\alpha_t^* = \underset{\alpha \in \{\text{Do Nothing, Replace}\}}{\arg\max} \; Q^{\pi^*}\left(\zeta_t, \alpha\right). \tag{15}$$

Based on this decision rule, the system's operation is terminated when the optimal Q-function suggests replacement or when the system fails, whichever occurs first.

**Algorithm 2.** Real-time Control Framework for the $j$th System

```
Input: Trained agent with optimal $Q$-function $Q^{\pi^*}(\zeta, \alpha)$.
Input: Initialize replacement cost $C = 0$ and $t = 0$.
Start: System Monitoring
- Set $t = t + 1$.
- Calculate the belief state $\zeta_t$ from available sensor data.
- Calculate two possible action-value function values and find the optimal action
    based on Eq. (15)
if $\alpha_t$ = "Replace" then
        $C = \frac{c_r}{t}$,
end if
if system has failed already then
        $C = \frac{c_r + c_f}{T_j}$
end if
Termination: If $C > 0$, then terminate the algorithm and output $t$ and $C$, otherwise go
    back to the Start.
Output: Total replacement cost $C$ and the effective replacement time $T^* = t$.
```

## 4.3. RL for warning generation and RUL estimation

The second Bayesian filtering-based DRL framework proposed in this paper introduces an original decision making approach to generate warnings and predict RULs for degrading systems. The first objective is to train a DRL agent that, given a time threshold $d$ defined by the user, generates an alarm for a working system when the age of the system is as close as possible to $d$ units before failure. The threshold $d$ can take any values less than $D$, where $D$ is a number that approximately represents the largest possible system's lifetime ($D \geqslant T_j$). The DRL agent should be trained to estimate a time window $R_d$ that closely approximates the difference between system failure time $T_j$ for system $j$ and time threshold $d$ at which the alarm should be generated. Since we have some training data with known lifetimes $T_{1:N}$, we can generate an agent that takes care of such a task. During the training phase, it is possible for the agent to generate early warning or late warning alarms. The algorithm's objective, however, is to train the agent to generate alarms as close as possible to $d$ units before system lifetime $T_j$. For that reason, we consider a reward function in which instant rewards are provided to the agent depending on the time an alarm (if any) is raised. More specifically, we define an action set $\mathcal{A}$ that contains two different actions, namely, "Continue" and "Warning". For the instant reward, we define an early warning cost/risk per time unit $c_e$, and a late warning cost/risk per time unit $c_l$, where $c_e < c_l$. The early warning and late warning costs/risks together represent the trade-off between generating the alarm earlier or later than the desired warning time. In other words, these two cost elements control the sensitivity of the model with regards to early and late predictions of failure with respect to an ideal threshold $d$, which is defined by the users. The ratio of early warning to late warning shows how important early warnings are with respect to late warnings. For critical systems with large failure/downtime costs and consequences, we should set the cost of late warning much larger than the cost of early warning. These two parameters are used as the inputs to train the DRL agent to choose between the two actions of warning and continue. Although the monetary value of these two cost elements may be explicitly known for some applications, the user of the model does not need the actual monetary values of such parameters. Instead the trade-off between these two elements can be quantified and used as inputs for the decision making frameworks. Also, their values can be fine-tuned to drive the policy that takes the desired actions.

To obtain a policy that minimizes the total warning cost for system $j$, given an ideal threshold $d$ and denoting the action $\alpha_t$ at time $t$, the reward function can be defined as follows:

$$r_t = \begin{cases} 0, & \alpha_t^j(d) = \text{“Continue”} \ \& \ t \leqslant T_j - d, \\ -c_e(T_j - d - t), & \alpha_t^j(d) = \text{“Warning”} \ \& \ t < T_j - d, \\ 0, & \alpha_t^j(d) = \text{“Warning”} \ \& \ t = T_j - d, \\ 0, & \alpha_t^j(d) = \text{“Continue”} \ \& \ t > T_j - d \ \& \ t < T_j, \\ -c_l(d - T_j + t), & \alpha_t^j(d) = \text{“Warning”} \ \& \ t > T_j - d \ \& \ t < T_j, \\ -c_l d, & t = T_j \text{ (regardless of action } \alpha_t^j(d)). \end{cases} \tag{16}$$

The reward function is based on three scenarios in which the agent can be found during the training phase. The first scenario is at $t \leqslant T_j - d$, at which the agent either receives no instant penalty (negative reward) if it chooses action "Continue", or it receives a penalty for early warning if it chooses to generate such a warning. Similarly, for $T_j - d < t < T_j$, if action "Continue" is chosen, the agent receives no penalty, otherwise a late warning penalty is considered. Both of these occasions encourage the agent to learn a policy for generating alarms as close as possible to $T_j - d$. The last scenario occurs at system failure when regardless of the action taken, the agent receives the highest instant penalty for late warning. Here, users can add more penalty terms for very early/late warning or warning at failure. Algorithm 1 can now be applied for every value of $d$ so that the DRL agent is trained at the end of the training phase. The trained agent returns $R^j(d)$ for system $j$ in which $R^j(d)$ tends to be as close as possible to $|T_j - d|$ depending on the trade-off between early and late warning costs. The warning time $R^j(d)$ can be found as

$$R^j(d) = \inf\{t: \alpha_t^j(d) = \text{Warning}\}. \tag{17}$$

In order to be able to use the above framework to estimate the remaining useful life, the process defined above needs to be repeated for all $d \in \{1, \ldots, D\}$. At any time point $t$, the output from each trained agent determines whether to continue (do nothing) or generate a warning. It should be pointed out that for predicting the remaining life at time $t$, we can only estimate the belief state up to time $t$. If there is at least one agent that suggests the action of a warning, then its threshold can determine the remaining life of the agent at time $t$. For example, at time $t$, if the agent with $d = 10$ suggests generating a warning, then this implies that the system is likely to fail in 10 cycles, that is, the remaining life is estimated to be 10 cycles. If multiple agents suggest the action of a warning, the one with the lowest $d$ determines the estimated remaining useful life. If no agent suggests warning, then the remaining life at that point cannot be determined. To avoid such a scenario, we can consider a larger number for $D$ so that at least one agent always suggests warning at any time point for a potentially large $d$. Algorithm 3-(a) provides an overview of the proposed procedure to train $D$ agents for warning generation and then Algorithm 3-(b) presents how the results from Algorithm 3-(a) can be used to estimate the remaining life at time $t$ for the $j$th system.

**Algorithm 3.** Steps to Estimate the Remaining Useful Life at Time $t$ for the $j$th System

```
Input: Training set, which includes N degrading systems with lifetimes
    $T_j, j \in \{1, 2, ..., N\}$.
Input: Set the number of threshold values $\{1, ..., D\}$.
Input: Belief states $\zeta^j_{1:t}, j \in \{1, ..., N\}$.
Input: Set costs $c_e$ and $c_l$.
Input: Initialize warning cost $C$.
(a) DRL agents training:
for $d = 1$ to $D$ do
    - Run Algorithm 1 based on the reward function in Eq. (16).
    - Get the trained agent and its associated optimal Q-function.
    - Determine whether to issue a warning based on the optimal Q-function.
end for
(b): Remaining Useful Life Estimation for System j at time t ($l_t^j$):
Input: The history of sensor data for system j up to time t
    - Calculate $\alpha_t^j(d)$ for all $d \in \{1, ..., D\}$ using the D trained agent from part (a)
Output: Calculate the remaining life $l_t^j$ as
$l_t^j = \min(d^*)$,  where $d^* = \{d; \alpha_t^j(d) = \text{Warning}\}$
```

It is important to note that for the warning generation process, at any time point $t$, the output from the trained agents determines whether to continue (do nothing) or generate/issue a warning. Later, the set of warnings help estimate the remaining life of the system as discussed in Algorithm 3. This means that there is no direct action on the operation of the system and only warnings are issued, which together can determine the remaining useful life as discussed in Algorithm 3. The warning itself may be further used to determine/initiate necessary maintenance actions (e.g., maintenance setup and order parts) before failure. Also, one may define a replacement policy directly based on the estimated RUL, that is, terminate the operation and replace the system if the estimated RUL is less than a predefined threshold.

Fig. 4 provides a high-level overview of the agent training process for maintenance decision making. The proposed frameworks rely only on sensor observations as inputs for decision making. The system is assumed to follow a generic and multi-layer hybrid state space structure that characterizes the stochastic relationship between latent states and sensor observations. Once the framework is trained from past data, then for any new system and at any time point, we can use sensor observation to estimate the probability distribution of latent states. Such a probability distribution (belief state) will be transformed into the inputs of reinforcement learning. Then the reinforcement leaning framework is trained using neural network structures as the Q-function approximator. Once the reinforcement learning agent is trained, it can employ direct sensor observations and transform them into optimal actions at any decision epoch. The reinforcement learning agents can be trained for three types of maintenance actions: when to replace a degraded system, when to generate warning signals based on the number of cycles left to failure, and predict the remaining useful life anytime during the system's operation.

### 4.4. An example of the application of the proposed frameworks

To better summarize the potential outcome of the proposed frameworks, we present a simple example from the CMAPSS dataset in Fig. 5 on the determination of optimal replacement time, the warning generation process, and remaining useful life estimation. The age of the selected system is $T = 54$ while the effective age (replacement time $T^*$) determined by the framework is 52. That is, the system's operation is suggested to terminate 2 cycles before the true failure point. The proposed framework for warning generation was repeated for $d \in \{45, 35, 12, 7\}$. It can be observed that for $d = 45$, the warning was issued at time $t = 10$, which is exactly 44 cycles before the failure point. For $d = 35$, $d = 12$, and $d = 7$, the warning was generated at times $t = 20$, $t = 42$, and $t = 47$, respectively, which gives $R(35) = 20$, $R(12) = 42$, and $R(7) = 47$. Based on these warnings, the operators may decide to initiate necessary maintenance setup actions (e.g., ordering parts). It can also be seen that the remaining useful life estimates (dashed line) are relatively close to the actual remaining life, particularly near the end of the lifecycle. This is mainly because more sensor data are employed for estimation.

## 5. Numerical experiments

We provide numerical experiments for both a simulated system and the CMAPSS turbofan engine degradation dataset from the NASA diagnostics repository (Saxena & Goebel, 2008). We first describe each data set and then provide results for the decision making framework and the RUL framework. It should be noted that for both datasets, we used the hybrid particle filter described in Section 3.2 and Algorithm 4 to obtain the particles, which provide latent state approximation for all systems in the training and validation datasets. Finally, we employed the process demonstrated in Section 4.1 to construct the final set of belief states that were later used as inputs for the RL frameworks.

### 5.1. Datasets description

#### 5.1.1. Simulation dataset

A fully stochastic framework was simulated for model evaluation. We generated $K = 200$ run-to-failure samples for a single-unit degrading system based on the hybrid structure presented in Eqs. (3)–(7). We assumed a generalized one-dimensional hidden degradation process $x_t$, a binary operating mode process $c_t$, a hazard process $\lambda_t$, a one-dimensional operating input $\boldsymbol{u}_t$, a two-dimensional observation process $\boldsymbol{y}_t$, and a working status process $o_t$. The structure of the system's dynamics is presented in Eqs. (18)–(20), (20)–(22) below:

$$p(c_t = k'|c_{t-1} = k) = p_{kk'}, \tag{18}$$

$$x_t = x_{t-1} - e^{(-b_{c_t})} - g \cdot u_t + \mathcal{N}(0, Q), \tag{19}$$

$$\lambda_t = \frac{1}{1 + e^{-(\alpha x_t + \beta_0)}}, \tag{20}$$

$$y_t = H^{c_t} x_t + \mathcal{N}(0, R), \tag{21}$$

$$\Pr\left(o_t = o\right) = \begin{cases} 1 - \lambda_t, & \text{if } o = 0 \\ \lambda_t, & \text{if } o = 1, \end{cases} \tag{22}$$

where $H$ is a $2 \times 1$ vector that maps the latent degradation state to sensor observations. Parameters $b$ and $g$ characterize the degradation evolution, and parameters $a$ and $\beta_0$ characterize the hazard process. A Markov transition probability matrix $p_{kk'}$ denotes the transition probabilities from state $i$ to state $k$. We consider two operating modes, namely, normal ($c_t = 0$) and faulty ($c_t = 1$). We also assume that as soon as a system transits to the faulty operating condition (mode), it remains

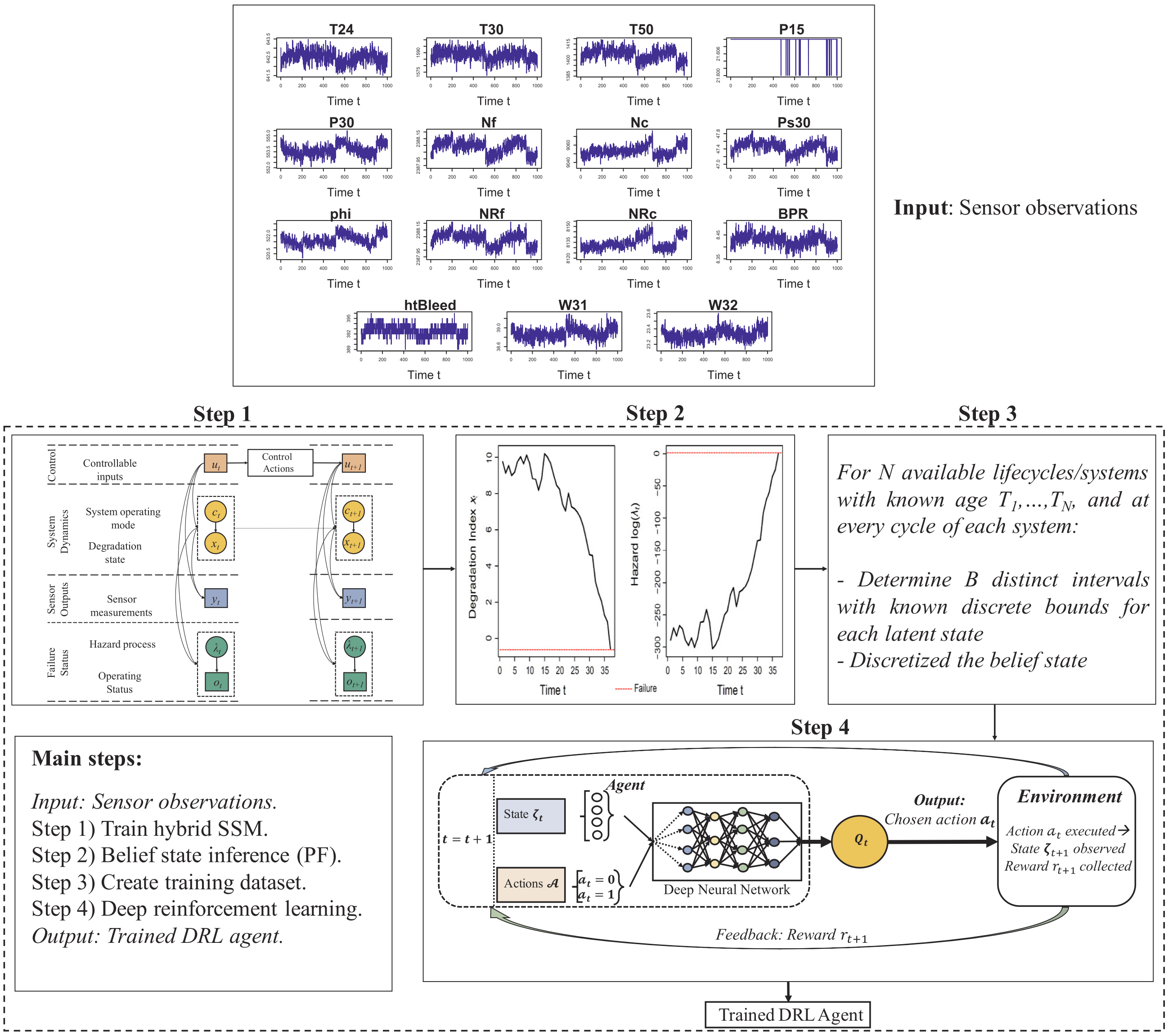
T24
T30
T50
P15
P30
Nf
Nc
Ps30
phi
NRf
NRc
BPR
htBleed
W31
W32
Time t
Input: Sensor observations
Step 1
Step 2
Step 3
Control
Controllable inputs
Control Actions
System Dynamics
System operating mode
Degradation state
Sensor Outputs
Sensor measurements
Failure Status
Hazard process
Operating Status
Degradation Index $x_t$
Hazard $\log(\lambda_t)$
Time t
Failure
For N available lifecycles/systems with known age $T_1,\ldots,T_N$, and at every cycle of each system:
- Determine B distinct intervals with known discrete bounds for each latent state
- Discretized the belief state
Step 4
Main steps:
Input: Sensor observations.
Step 1) Train hybrid SSM.
Step 2) Belief state inference (PF).
Step 3) Create training dataset.
Step 4) Deep reinforcement learning.
Output: Trained DRL agent.
Agent
State $\zeta_t$
Actions $\mathcal{A}$
$a_t = 0$
$a_t = 1$
$t = t + 1$
Deep Neural Network
$Q_t$
Output:
Chosen action $a_t$
Environment
Action $a_t$ executed→
State $\zeta_{t+1}$ observed
Reward $r_{t+1}$ collected
Feedback: Reward $r_{t+1}$
Trained DRL Agent

**Fig. 4.** An overview of the proposed framework and steps needed for agent training with sample sensor observations as inputs.

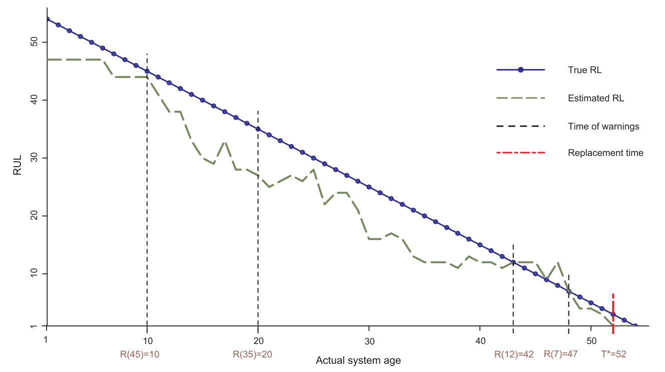

**Fig. 5.** Overview of the proposed frameworks with 3 potential outcomes: warning times (estimated for 3 values for $d$), replacement time, and remaining life estimates (throughout the lifecycle).

there for the rest of its lifetime (the same assumption made in the CMAPSS dataset). The transition matrix in Eq. (18) has the following form:

$$\mathbf{P} = \begin{bmatrix} p_{11} & 1 - p_{11} \\ 0 & 1 \end{bmatrix}. \tag{23}$$

Degradation and sensor observation processes are assumed to be contaminated by normally distributed noise with zero mean, variance $Q$, and covariance $R$. The true parameter values chosen for simulation are as follows:

$$b_0 = 6,\ b_1 = 2,\ g = 0.01,\ H^0 = \begin{bmatrix} 2.5 \\ 2.72 \end{bmatrix},\ H^1 = \begin{bmatrix} 1.76 \\ 1.09 \end{bmatrix}$$

$$p_{11} = 0.9,\ Q = 0.1,\ \alpha = -30,\ \beta_0 = 3,\ \mathbf{R} = \begin{bmatrix} 0.3 & 0 \\ 0 & 0.3 \end{bmatrix}.$$

The simulation process starts from initial values that denote an as-good-as-new state, where $c_1 = 0$ and $x_1 = 10$, and continues as follows: at cycle $t$, mode $c_t$, and degradation state $x_t$, the sensor vector $y_t$, and the hazard value $\lambda_t$ are generated according to their stochastic distributions. Then, a random number $\omega \sim \mathcal{U}(0, 1)$ was generated to compare with $\lambda_t$. If $\omega \leqslant \lambda_t$, then the system is considered failed ($o_t = 1$), otherwise $o_t = 0$ and the process continues. We performed the same procedure multiple times to obtain multiple independent time series.

### 5.1.2. CMAPSS dataset

The proposed framework was also evaluated on the CMAPSS turbofan engine degradation dataset provided by the NASA Ames Research Center (Saxena, Goebel, Simon, & Eklund, 2008). In total, there are 21 different observation variables ($y$) and three-dimensional operating inputs ($u$). Each engine starts its operation with an unknown level of initial wear and manufacturing variation implying the randomness of the engine's degradation. All engines were assumed to suffer from high-pressure compressor degradation. The HSSM presented in Eqs. (18)–(20), (20)–(22) was utilized to describe the latent dynamics of the turbofan engine dataset. The observation process presented in Eq. (21) was trained through the ELM neural network. The model parameters ($\mathbf{\Theta} = \{\theta_c, \theta_x, \theta_\lambda\}$) are $\theta_c = \{p_{11}\}$, $\theta_x = \{b_0, b_1, g_1, g_2, g_3\}$, and $\theta_\lambda = \{\alpha, \beta_0\}$. Here, parameter $g$ becomes a vector in order to accommodate the three-dimensional operating input process $u$. Parameter values are estimated as $\mathbf{\Theta} = \{0.899, 3.48, 1.99, 0.012, 0.022, 0.052, -24.8, 4.05\}$.

## 5.2. Structure of the neural networks

We tested our proposed methods using 3 neural network structures for Q-function approximators in order to find the best network that represents a proper mapping between states and actions. These structures are listed below: (a) A single-hidden layer neural network with 50 neurons in the hidden layer, (b) A fully connected deep neural network with two hidden layers of 32 and 16 neurons, and (c) A deep recurrent neural network of two hidden layers with 64 and 32 memory cells, respectively. Stacking multiple hidden layers in RNNs adds levels of abstraction of input observations over time but increases the CPU time for training. The number of hidden neurons and hidden layers for the DNN architecture was determined with cross-validation to better represent the interactions between the input data and their target values, given the limitations of each architecture. The rectified linear unit (ReLU) was used as the activation function between all layers except for the last hidden layer to the output layer where a linear activation function was used to generate the Q-function values. For the optimizer, we used RMSprop as one of the most well-known optimizers for deep learning. The RMSprop's hyperparameters were kept to their default values (i.e., the learning rate of $\lambda = 0.001$ and float of $\rho = 0.9$). The mathematical forms of the above three network structures are summarized in Appendix B.

## 5.3. Real-time decision making with reinforcement learning

To demonstrate the effectiveness of the proposed real-time framework for maintenance decision making, we first define a set of benchmark policies for comparison purposes. These policies are (a) an *ideal maintenance policy*, (b) a *corrective maintenance policy*, and (c) a *time-based maintenance policy* or a *preventive maintenance policy*. The optimal cost associated with each of these benchmark policies can be computed as follows:

(a) *Ideal maintenance cost (IMC)*: Based on this hypothetical policy, the system's operation is terminated exactly one cycle before failure so that the system useful life is maximized and system failure is prevented. The expected cost associated with this policy can be computed as

$$\phi_{IMC} \approx \frac{N \cdot c_r}{N \cdot (\mathbb{E}(T) - 1)} \approx \frac{N \cdot c_r}{\sum_{j=1}^{N} (T_j - 1)}, \tag{24}$$

where $T_j,\ j \in \{1, 2, \ldots, N\}$ denotes the lifetimes of the $j$th system. This cost represents the optimal return value of the reward function, that is, replacements always occur at $t = T_j - 1$ for all $j \in \{1, \ldots, N\}$. In theory, no other policy can provide a better cost than Eq. (24), which is why it can be a great reference for the comparison. To our knowledge, no policy actually exists that can provide such a low cost. We used it to be able to evaluate how close the obtained cost from our policy is to this lowest limit.

(b) *Corrective maintenance cost (CMC)*: Based on this policy, the maintenance starts right after the system fails. Thus, each system always operates up to its highest useful life but is also always subject to the failure cost. The expected cost of this policy, which can be considered as the upper bound for any policy, can be computed as follows:

$$\phi_{CMC} \approx \frac{(c_r + c_f)}{\mathbb{E}(T)} \approx \frac{N \cdot (c_r + c_f)}{\sum_{j=1}^{N} T_j}. \tag{25}$$

(c) *Time-based maintenance cost (TBMC)*: The objective of this policy is to find the time threshold $T^*$ when the system needs to be replaced regardless of its status. In other words, the system is replaced at time $T^*$ or at failure, whichever occurs first. To numerically find $T^*$, we calculate the empirical average maintenance cost for a time horizon $t = \{1, 2, \ldots, D\}$, which $t = 1$ denotes the time each system starts functioning and $D$ is the maximum possible lifetime in the historical data. Using exhaustive search, we obtain $T^*$ where the average unit cost is minimized. Thus, we have

$$T^* = \underset{t \in \{1, \ldots, D\}}{\arg\min} \frac{1}{N} \sum_{j=1}^{N} \left[ \mathbf{1}_{\{t < T_j\}} \frac{c_r}{t} + \mathbf{1}_{\{t \geqslant T_j\}} \frac{c_r + c_f}{T_j} \right], \tag{26}$$

$$\phi_{TBMC} = \frac{1}{N} \sum_{j=1}^{N} \left[ \mathbf{1}_{\{T^* < T_j\}} \frac{c_r}{T^*} + \mathbf{1}_{\{T^* \geqslant T_j\}} \frac{c_r + c_f}{T_j} \right], \tag{27}$$

where the first part in the summation in Eq. (26) is for the cases in which the system's age is larger than $t$, and the second part is for cases in which the system age is less than $t$. In all numerical experiments, we fixed the cost of replacement to be $c_r = 100$ and considered five different values for the failure cost $c_f$, that is we set $c_f \in \{25, 50, 100, 500, 1000\}$. The proposed framework was tested for the three neural network architectures discussed in Section 3.4.

### 5.3.1. Results for the simulated dataset

We used $N = 150$ simulated systems for training and $\widehat{N} = 50$ for testing. We first evaluated the convergence of the DRL framework. Outcomes from $Q$-functions using different values of $c_f$ are shown in Fig. 6. It can be seen from this figure that the agent reaches an optimal state after some iterations for all three networks and almost all cost combinations. To better observe the approximation error and convergence, we plotted the root mean squared error (RMSE) of the neural network loss over different episodes in Fig. 7. This figure shows that the neural networks used in the proposed DRL policy perform reasonably well in terms of estimating $Q$, and the RMSE values converge to a small value. Among all network structures, the deep neural network performs the best in terms of convergence.

After training each agent, Algorithm 2 was implemented for estimating the replacement time of each system and calculating the average replacement cost of the systems within the test set $\widehat{N}$. Table 1 presents the results compared with the benchmark models discussed in Section 5.3.1. The first result is for the average replacement cost for five

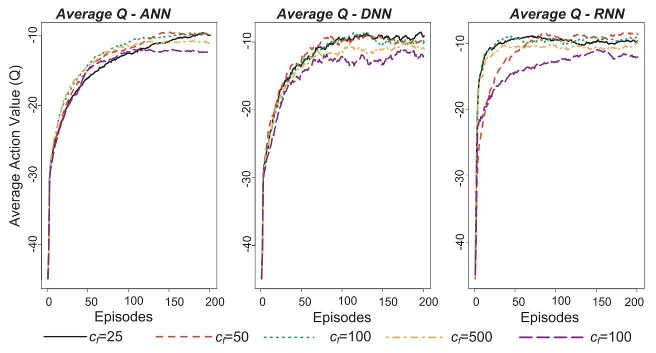

**Fig. 6.** Evolution of the $Q$-function during the training phase.

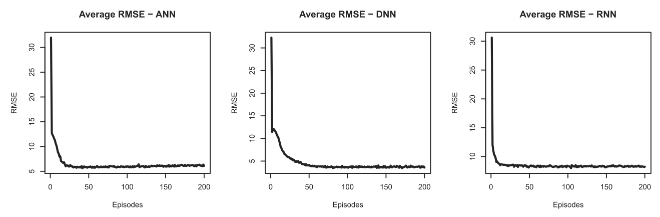

**Fig. 7.** Average neural network loss for three network structures ANN, DNN, and RNN during the training phase.

**Table 1**
Average replacement costs, time, and failure rate for $\hat{N}$ (simulated data).

| | $c_f$ | $c_r$ | ANN | DNN | RNN | IMC | TBMC | CMC |
|---|---|---|---|---|---|---|---|---|
| Cost | 25 | 100 | 5.34 | 5.78 | 5.29 | 4.54 | 5.41 | 5.43 |
| | 50 | 100 | 5.48 | 6.13 | 6.07 | 4.54 | 6.48 | 6.52 |
| | 100 | 100 | 5.62 | 6.38 | 5.35 | 4.54 | 8.09 | 8.69 |
| | 500 | 100 | 6.49 | 6.17 | 5.81 | 4.54 | 10.02 | 26.06 |
| | 1000 | 100 | 5.59 | 6 | 5.34 | 4.54 | 10.93 | 47.79 |
| Time | 25 | 100 | 19.68 | 18.08 | 19.84 | 22.02 | 22.58 | 23.02 |
| | 50 | 100 | 19.52 | 18.58 | 20.26 | 22.02 | 19.28 | 23.02 |
| | 100 | 100 | 19.58 | 16.3 | 19.68 | 22.02 | 13.76 | 23.02 |
| | 500 | 100 | 18.5 | 17.82 | 15.92 | 22.02 | 10.98 | 23.02 |
| | 1000 | 100 | 17.9 | 16.66 | 17.9 | 22.02 | 10.98 | 23.02 |
| Failure | 25 | 100 | 20% | 28% | 46% | 0% | 94% | 100% |
| | 50 | 100 | 14% | 19% | 46% | 0% | 54% | 100% |
| | 100 | 100 | 10% | 4% | 20% | 0% | 12% | 100% |
| | 500 | 100 | 4% | 2% | 2% | 0% | 2% | 100% |
| | 1000 | 100 | 0% | 0% | 0% | 0% | 2% | 100% |

different combinations of $(c_r, c_f)$ and all three neural network architectures (ANN, DNN, RNN). Also, the average cost for the three benchmark policies is reported (IMC, TBMC, CMC). As expected, the ideal replacement policy IMC provides the lowest cost. Our model provides the second best results and performs better than the time-based and corrective policies.

### 5.3.2. *Results for the CMAPSS dataset*

The numerical experiments presented in Section 5.3.1 were also applied to the CMAPSS dataset in order to observe how well the replacement points can be obtained. The structures of the neural networks are the same as the simulation dataset. The dataset consists of 100 degradation time-series, out of which $N = 80$ were randomly chosen for training and $\hat{N} = 20$ were used for testing. The trained agent was then evaluated on the testing set $\hat{N}$, and the results obtained are presented in Table 2. As seen in this table, the ideal maintenance cost policy IMC provides the best results. However, our model performs reasonably well in terms of minimizing cost, preventing failure replacement, and maximizing system operation. Also, potentially due to their structures that are able to handle more complexity, DNN and RNN perform better than the regular ANN.

**Summary**. To better compare the performance of our neural network structures over TBMC and CMC, we conducted a set of statistical tests based on the differences between the cost obtained from our work and those from the two benchmark models of TBMC and CMC. The results are shown separately for the simulated and CMPASS datasets and each case of $c_f$ ($c_r$ is fixed at 100). For comparing every pair of two methods, since the sample populations (i.e., testing samples) are the same for the two methods (i.e., same samples are used by each method), a pairwise t-test was employed to test the significance of the difference between the average replacement costs. The null hypothesis of the t-test was that there was no difference between the average costs of the two models (i.e., our model performs similarly to the corresponding benchmark model). The alternative hypothesis was that the mean cost of our model is lower than the cost in the corresponding benchmark

**Table 2**
Average replacement costs, time, and failure rate for $\hat{N}$ (CMAPSS data).

| | $c_f$ | $c_r$ | ANN | DNN | RNN | IMC | TBMC | CMC |
|---|---|---|---|---|---|---|---|---|
| Cost | 25 | 100 | 2.48 | 2.13 | 2.15 | 1.93 | 2.23 | 2.37 |
| | 50 | 100 | 2.27 | 2.1 | 2.1 | 1.93 | 2.31 | 2.84 |
| | 100 | 100 | 2.32 | 2.09 | 2.02 | 1.93 | 2.38 | 3.78 |
| | 500 | 100 | 2.27 | 2.21 | 2.05 | 1.93 | 2.38 | 11.34 |
| | 1000 | 100 | 2.16 | 2.6 | 2.04 | 1.93 | 2.38 | 20.8 |
| Time | 25 | 100 | 40.4 | 47 | 49.9 | 51.9 | 46.6 | 52.9 |
| | 50 | 100 | 45.25 | 47.25 | 47.3 | 51.9 | 46.6 | 52.9 |
| | 100 | 100 | 43.05 | 47.8 | 48.45 | 51.9 | 42 | 52.9 |
| | 500 | 100 | 44.05 | 45.35 | 47.2 | 51.9 | 42 | 52.9 |
| | 1000 | 100 | 46.35 | 38.6 | 48.75 | 51.9 | 42 | 52.9 |
| Failure | 25 | 100 | 0% | 0% | 35% | 0% | 15% | 100% |
| | 50 | 100 | 5% | 0% | 5% | 0% | 15% | 100% |
| | 100 | 100 | 0% | 0% | 0% | 0% | 0% | 100% |
| | 500 | 100 | 0% | 0% | 0% | 0% | 0% | 100% |
| | 1000 | 100 | 0% | 0% | 0% | 0% | 0% | 100% |

model. The statistical tests were conducted with respect to a 5% level of significance. The p-values of the corresponding t-tests are shown in Table 3 for each pair of comparisons. It should be noted that p-values close to zero imply that the null hypothesis is rejected and that our model obtains lower replacement costs. Results shown in Table 3 verify that our model performs better than the benchmark models in many settings, particularly when the cost of failure is higher. It should be noted that IMC is not included in these statistical experiments because it is only a hypothetical policy that shows the lower bound of the replacement costs. Also, the statistical tests were conducted only on the average cost because it is the main decision making criterion in this paper.

The results shown in Tables 1,2 are summarized below. It can be seen from Tables 1,2 that as the cost of failure replacement increases, all models tend to suggest earlier replacement times, which results in lower ratios of failure replacements (i.e., more preventive replacements occur). For such cases, the agents in our model tend to terminate the operation earlier and prevent more failure replacements. It can also be seen that the corrective maintenance policy can result in the highest utilization of lifetime but it is subject to 100% failure replacements (i.e., all systems are allowed to fail). For systems with a failure cost close to zero, such a policy may still be acceptable. Because of the high rate of failure replacements in such a policy, the corresponding costs are often the highest among all models. The time-based maintenance policy has a reasonable performance for lower values of failure cost; however, it becomes very sensitive with larger values of $c_f$. For instance in the case of simulated data, when the cost of failure is 500, this policy suggests replacing 98% (i.e., 1–2%) of systems at early points during system's operation with the average effective replacement time of 10.98, which is about 50% smaller than the average operation time of all samples. Such a fixed-time policy may be useful only when either the replacement cost is close to zero or when all systems follow very similar degradation patterns with similar lifetimes. Overall, the three network structures perform better than all the benchmark models, and their performance is relatively close to the ideal case of IMC, which theoretically gives the lowest possible cost for the systems. Among all three structures, RNN performs slightly better. This is mainly because it has a more complex structure and can better capture the temporal behavior of the system's dynamics. The results of the statistical tests shown in Table 3 also verify that the three neural network structures perform better in most settings.

### 5.4. *Warning generation and RUL estimation*

The framework in Section 4.3 was validated using the same training and testing datasets in Sections 5.3.1 and 5.3.2. The maximum number

**Table 3**
P-values of the $t$-tests for the mean cost difference (our model - benchmark model).

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Simulated Dataset - Testing Set** | | | | | | | | | | |
| | TBMC | | | | | CMC | | | | |
| cf | 25 | 50 | 100 | 500 | 1000 | 25 | 50 | 100 | 500 | 1000 |
| ANN | 0.561 | 0.007 | * | 0.058 | 0.020 | 0.915 | 0.001 | * | * | * |
| DNN | 0.884 | 0.033 | * | 0.024 | 0.063 | 0.953 | 0.052 | 0.007 | * | * |
| RNN | 0.475 | 0.038 | * | 0.042 | 0.020 | 0.705 | 0.016 | 0.104 | * | * |
| **CMAPSS Dataset - Testing Set** | | | | | | | | | | |
| ANN | 0.999 | 0.631 | 0.543 | 0.427 | 0.088 | 0.916 | * | * | * | * |
| DNN | 0.470 | 0.037 | 0.017 | 0.219 | 0.188 | 0.017 | * | * | * | * |
| RNN | 0.506 | 0.147 | 0.017 | 0.016 | 0.013 | 0.002 | * | * | * | * |

$*$means less than $10^{-3}$.

of threshold values was set to $D = 50$; therefore, 50 RL agents were trained (i.e., $d \in \{1, ..., 50\}$) for every neural network architecture described in Section 3.4 using Algorithm 3-(a). The first outcome of the $d$th agent is to monitor the most updated set of sensor data over time to generate warning $d$ units before the actual failure points. After the training process was completed (see Algorithm 3-(a)), the agents were employed to estimate the RUL for the systems in the test sets. For a better comparison, we considered three cost combinations for early warning ($c_e$) and late warning ($c_l$) in the cost function presented in Eq. (16): (i) **C1:**$\{c_e = 100, c_l = 500\}$, (ii) **C2:**$\{c_e = 500, c_l = 500\}$, and (iii) **C3:**$\{c_e = 500, c_l = 100\}$.

### 5.4.1. *Results for the simulation dataset*

In Fig. 8, we compare true $d$s and their estimated values averaged over all systems in the training set for $d = \{1, 2, ..., 10\}$. It can be seen that the agents tend to overestimate $d$; they generate more early warnings, when the cost of early warning is lower than the cost of late warning ($c_e < c_l$). Also, the exact opposite occurs when the cost of early warning is higher than its late warning counterpart ($c_e > c_l$). The best case occurs when both costs are equal (middle column on Fig. 8).

To see how well the $d$th agent can predict system failure, we applied Algorithm 3-(b) to every system in the testing set $\hat{N}$ to check at what points the warnings are generated, and subsequently estimate their RULs. In Fig. 9, we provide a scatterplot that compares the true (which is known) and estimated RULs in terms of % of a true lifetime for all 50 systems in $\hat{N}$. We also show the average % values. As expected, the agents slightly underestimate RUL when $c_e < c_l$. However, in the case in which RNN is used for $Q$-function approximator, the RUL estimates closely approximate their true values. In the case in which $c_e > c_l$, the RUL estimates for all different $Q$-function approximators are overestimated, which is reasonable since the agents are more likely to generate late warnings. Finally, when $c_e = c_l$, we observe a slight RUL overestimation during the first stages of system operation. Nevertheless, when the RNN is used for $Q$-function approximator, we observe a convergence between true and estimated RULs during the last 10% of system lifetime.

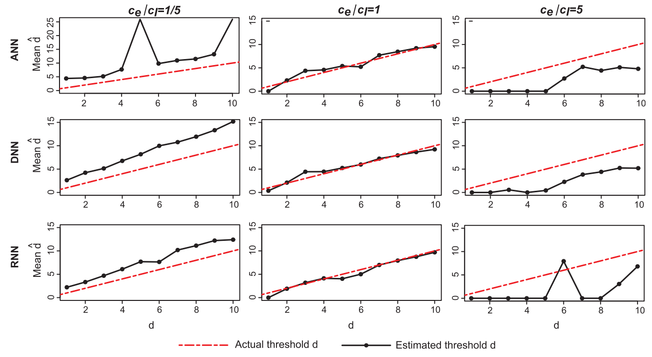

**Fig. 8.** Comparison between the true $d$s and their estimates for all systems in the training set and for every cost combination.

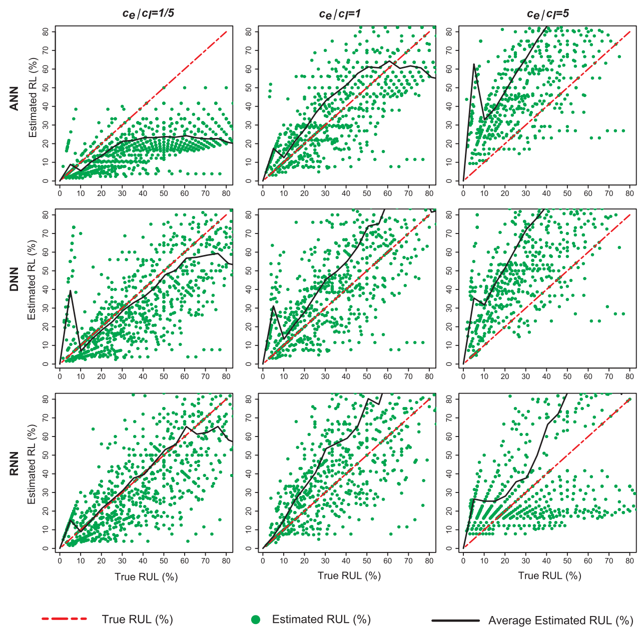

**Fig. 9.** Distribution of RUL estimates as percentages of true lifetimes for three cost combinations. The dots present the estimate for each system.

### 5.4.2. *Results for the CMAPSS dataset*

The framework was also tested on the CMAPSS dataset. Similar to Section 5.4.1, we trained $D = 50$ agents using Algorithm 3-(a). The comparison between true $d$ and estimated $\hat{d}$ values, averaged over all systems on the training set $N$, is presented in Fig. 10. Again, the estimated values $\hat{d}$ converge closer to the true values $d$ for cost combination **C2** ($c_e = c_l$). Similar to the simulated data, we applied Algorithm 3-(b) on every system in the testing set $\hat{N}$ to observe the points at which warnings are generated. The results of comparing the true and estimated RULs in terms of percentage for all 20 systems in $\hat{N}$ are shown in Fig. 11, along with the average % values. The plots shown verify the reasonable performance of the proposed RUL estimation framework. In the case in which $c_e = c_l$, the average estimated RUL almost equals the true RUL, especially closer to the actual system failure. Furthermore, when $c_e < c_l$, the agents suggest earlier replacement times, whereas the opposite occurs when $c_e > c_l$.

**Summary**. In summary, the results for both simulated and CMAPSS

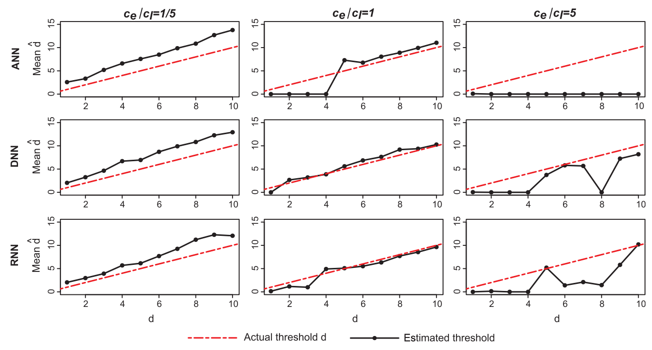

**Fig. 10.** Comparison between true and average warning thresholds for all systems in the validation set $\hat{N}$ and for all cost combinations.

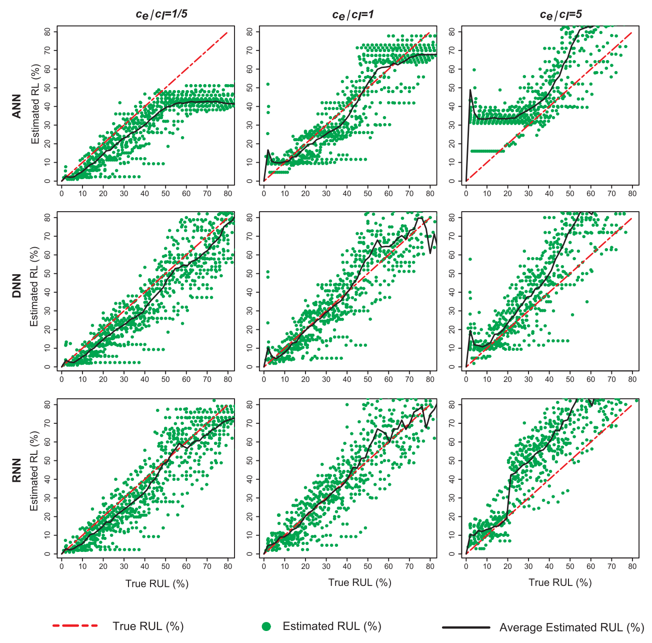

**Fig. 11.** Distribution of RUL estimates as % of true lifetimes for three cost combinations.

datasets imply that the proposed frameworks can provide a reliable way to generate warnings and estimate RULs at different time points while the system is operating. At any time point, our frameworks utilize the most updated set of sensor data to generate warnings and predict the remaining life. We can observe from the results that the best RUL estimates are calculated when the costs of early and late warnings are equal. The users of our models have the opportunity to change the trade-off between early and late warnings and adjust the estimation results. With regards to the neural network structures used as function approximators, we can see that RNN and DNN perform better than the typical ANN. One reason is the fact that these structures can accommodate more complex environments.

### 5.5. Computational time

In this subsection, we numerically analyze the scalability of the proposed frameworks with experiments that are conducted for different sizes of simulated training sets (i.e., 150, 800, 3000, and 8000 samples) and for 200 episodes. It is expected that ANN and DNN require significantly less computational time for agent training compared to RNN because it needs to sequentially analyze the input data. In Table 4, we present the results for the two frameworks discussed in Sections 4.2 and 4.3. As expected, ANN and DNN generally perform well in terms of CPU time even when the training set becomes very large. However, the CPU time for RNN training increases significantly with the number of samples. This outcome implies that while RNNs perform well as a $Q$-function approximator, the training time can potentially be a major challenge. Similar to the maintenance cost framework, ANN and DNN require much less training time, regardless of the size of the training data. On the other hand, the CPU time for RNN increases significantly. Although RNN gives the most accurate RUL estimation results, the user has to carefully consider the trade-off between training time and model accuracy.

**Table 4**
CPU time (seconds × 100) comparisons for the frameworks described in Sections 4.2 and 4.3.

| Framework | Network Structure | Training Set Size | | | |
|---|---|---|---|---|---|
| | | 150 | 800 | 3,000 | 8,000 |
| Framework 1 (Section 4.2) | ANN | 3 | 32 | 84 | 232 |
| | DNN | 4 | 34 | 86 | 234 |
| | RNN | 60 | 450 | 1614 | 4302 |
| Framework 2 (Section 4.3) | ANN | 1 | 8 | 21 | 58 |
| | DNN | 1.25 | 8.5 | 21.5 | 58.5 |
| | RNN | 23.2 | 112.5 | 403.5 | 1075.5 |

## 6. Summary and future work

This article develops a new generation of dynamic decision frameworks inspired by DRL to transform real-time sensor data collected from monitoring sensors into decision-making intelligence and actionable insights. In this paper, we proposed two original frameworks for degrading systems using DRL with stochastic environments: (i) a real-time control and decision making framework for maintenance replacement and (ii) an RUL estimation framework that can also be used to generate warnings at any point while the system is operating. Bayesian filtering was utilized to infer system latent dynamics and transform sensor data to RL inputs. In future work, we will investigate fully nonparametric models to characterize system dynamics and develop RL frameworks. We will also study how the framework can be used for multi-unit systems with competing and non-competing failure modes.

## CRediT authorship contribution statement

**Erotokritos Skordilis:** Methodology, Writing, Validation. **Ramin Moghaddass:** Methodology, Writing, Supervision, Validation.

**Acknowledgement**

This article is partially supported by National Sciene Foundation grant number 1846975.

## Appendix A

The hybrid particle filter described in Section 3.2 is presented in Algorithm 4.

**Algorithm 4.** Hybrid Particle Filter

```
Input:
```

```
Sequences of observations $y_{1:T}$, $o_{1:T}$, parameter vector $\boldsymbol{\Theta} = \{\theta_c, \theta_x, \theta_\lambda, \theta_y, \theta_o\}$, number of particles $N_s$.
for $t = 1$ to $T$ do
  for $i = 1$ to $N_s$ do
    Sample $c_t$, $x_t$, $\lambda_t$ from the following distributions:
      $c_t^i \sim p(c_t|c_{t-1}^i, u_t, \theta_c)$,
      $x_t^i \sim p(x_t|x_{t-1}^i, c_t^i, u_t, \theta_x)$,
      $\lambda_t^i \sim p(\lambda_t|x_t^i, c_t^i, u_t, \theta_\lambda)$.
  end for
  for $i = 1$ to $N_s$ do
    Calculate the importance weights as:
      $w_t^i \propto w_{t-1}^i \frac{p(y_t \mid x_t^i, c_t^i) p(o_t \mid \lambda_t^i) p(x_t^i \mid x_{t-1}^i, c_t^i) p(c_t^i \mid c_{t-1}^i) p(\lambda_t^i \mid x_t^i, c_{1:t}^i)}{q(x_t^i \mid x_{1:t-1}^i, c_{1:t}^i, y_{1:t}) q(c_t^i \mid c_{1:t-1}^i, y_{1:t}) q(\lambda_t^i \mid x_{1:t}^i, c_{1:t}^i, o_{1:t})}$.
    Normalize the importance weights:
      $\widetilde{w}_t^i = \frac{w_t^i}{\sum_{j=1}^{N_S} w_t^j}$.
  end for
  for $i' = 1$ to $N_s$ do
    - Draw new particles $c_t^{i'}$, $x_t^{i'}$, $\lambda_t^{i'}$ with replacement such that:
    $p(c_t^{i'} = c_t^i) = \widetilde{w}_t^i, \quad i' \in \{1, ..., N_s\}$,
    $p(x_t^{i'} = x_t^i) = \widetilde{w}_t^i, \quad i' \in \{1, ..., N_s\}$,
    $p(\lambda_t^{i'} = \lambda_t^i) = \widetilde{w}_t^i, \quad i' \in \{1, ..., N_s\}$,
    Set $w_t^i = \frac{1}{N_S}, \quad i \in \{1, ..., N_s\}$.
  end for
end for
Output: Particles $c_{1:T}^i$, $x_{1:T}^i$, $\lambda_{1:T}^i$, and weights $w_{1:T}^i$ for $i \in \{1, ..., N_s\}$.
```

## Appendix B

In this appendix, the structures of the three neural networks used for characterizing the Q-Function are discussed.

### B.1. Single hidden layer neural network

*Artificial neural networks* (ANNs) consist of a collection of fully connected units called *artificial neurons* distributed in three sequential, densely connected layers, namely, input, hidden, and output layers. Neurons in each layer collect input signals and produce outputs by utilizing a nonlinear *activation function* over the sum of all inputs. At first, assuming a *d-dimensional* input vector $\xi^d = \{\boldsymbol{\zeta}, \alpha\}$ that contains a belief state and an action, a *n-dimensional* hidden layer, a *m-dimensional* output layer, and an activation function $h(\cdot)$, the neurons of the hidden layer are activated as:

$$l_j = h\left(\sum_i^d w_{ij}\xi^i + w_{0j}\right), \forall j \in \left\{1, 2, ..., n\right\},$$

where $w_{ij}$ denotes the weights of the links between the input and hidden layers, and $w_{0j}$ denotes the hidden layer *bias*. The activation function is generally chosen to be sigmoidal, with *logistic sigmoid* $\sigma(\cdot)$ to be the most popular choice (Bishop, 2006):

$$h(v) \equiv \sigma(v) = 1/(1 + \exp(-v)).$$

Following the same process, output layer neurons are activated by activation function $f(\cdot)$ and linear combinations of hidden neuron values as $\hat{y}_k = f\left(\sum_j^n w_{jk} l_j + w_{0k}\right), \forall k \in \left\{1, 2, ..., m\right\}$, where $w_{jk}$ denotes the connection weights between the hidden and output levels, and $w_{0k}$ denotes the

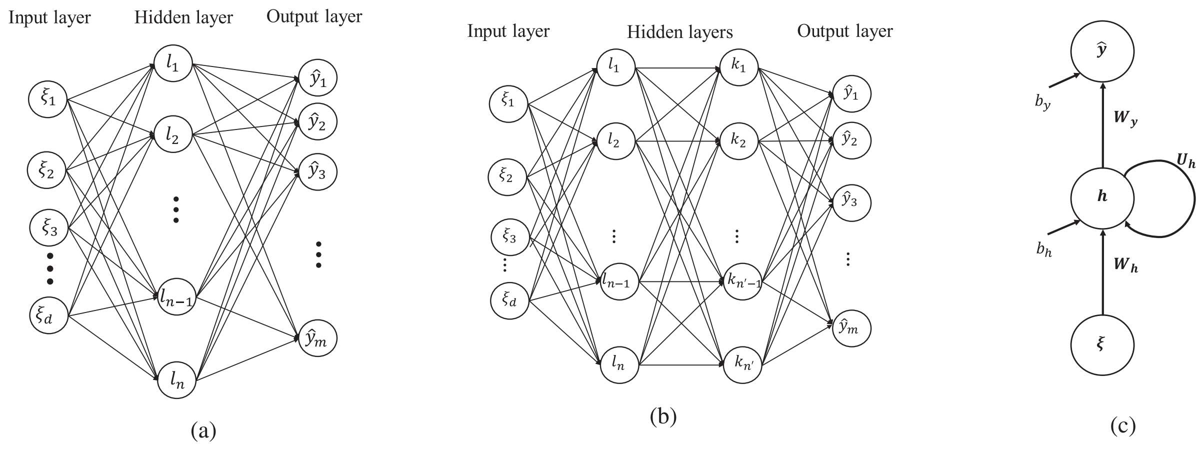

**Fig. 12.** The structures of the Neural network architectures (a) ANN, (b) DNN, (c) RNN.

output layer bias. Network training is achieved through the *backpropagation* algorithm. Fig. 12(a) gives an overview of a simple ANN.

### *B.2. Deep fully connected neural network*

Fully connected deep neural networks (DNNs) follow the same principles as ANNs. The only difference between ANNs and DNNs is that DNNs have at least two hidden layers. DNNs can be more advantageous than ANNs since the increased number of hidden layers gives DNNs the potential of approximating more complex functions. Different layers can have different numbers of neurons. A common practice in deep learning reduces the number of neurons between successive hidden layers, which leads to better generalization and less overfitting. However, it can also lead to underfitting, since dropped neurons contain information stored in the training data. Another issue is that multiple hidden layers contain a large number of hidden neurons, thus more free variables (weights) need to be tuned through backpropagation. Fig. 12(b) presents an outline of a DNN with two hidden layers.

### *B.3. Deep recurrent neural network*

Deep recurrent neural networks (DRNNs) are deep neural networks specifically designed for processing sequential data. In DRNNs, connections between units form directed sequential graphs that allow them to display temporal dynamic behavior. The main difference compared to dense neural networks is that the latter can only map given inputs to target outputs using only forward propagation. On the other hand, DRNNs are able to map the entire input history to the target outputs by allowing the memory of all these inputs to remain stored within the network (Zhao, Yan, et al., 2019). Many different DRNN architectures have been introduced in the literature. A simple DRNN takes as input a sequence of data points and at each time step a simple DRNN unit is applied to a single data point as well as to the network's output from the previous time step (Gal, 2016). Given an input in the form of $\boldsymbol{\xi}_{1:d}^{1}, \ldots, \boldsymbol{\xi}_{1:d}^{\tau}$, where $\boldsymbol{\xi}_{1:d}^{1:\tau}$ denotes the $d \times \tau$ tensor representing $d$-dimensional input sequences at times $t = 1, ..,\tau$, a simple DRNN applies an activation function $f_h(\cdot)$ iteratively. This process creates a latent level $\mathbf{h}_t$ as follows:

$$h_t = f_h(\boldsymbol{\xi}_t, h_{t-1}) = \sigma_h(\boldsymbol{\xi}_t \cdot \mathbf{W}_h + h_{t-1} \cdot U_h + b_h),$$

where $W_h$, $U_h$ are neuron weight matrices associated with the inputs at time $t$ and the feedback outputs from time $t-1$; respectively, $b_h$ is the bias; and $\sigma(\cdot)$ represents the logistic sigmoid activation function. The output at time $t$ can then be defined as:

$$\hat{y}_t = f_{\hat{y}}(h_t \cdot W_{\hat{y}} + b_{\hat{y}}) = \sigma_{\hat{y}}(h_t \cdot W_{\hat{y}} + b_{\hat{y}}),$$

where again $W_{\hat{y}}$ denotes a neuron weight matrix associated with the outputs at time $t$ and $b_{\hat{y}}$ the output bias. Fig. 12(c) presents the layout of an RNN.

It is known that due to the dynamic nature of typical RNNs, such as Vanilla RNNs, where the gradients are being propagated back in time up to the initial layer, they are subject to the problem of vanishing/exploding gradient (Schmidhuber, 2015). The two popular and widely used methods of long short-term memory (LSTM) (Hochreiter & Schmidhuber, 1997) and Gate Recurrent Unit (GRU) (Chung, Gulcehre, Cho, & Bengio, 2014) have the potential to effectively keep long-term dependencies in sequence data while handling the vanishing/exploding gradient issues in Vanilla RNNS. In this paper, we employed GRU, which is faster to train and has less parameters compared to LSTM since it does not have to use a memory unit to control the flow of information.

## References

Allen, T. T., Roychowdhury, S., & Liu, E. (2018). Reward-based monte carlo-bayesian reinforcement learning for cyber preventive maintenance. *Computers & Industrial Engineering, 126*, 578–594.

Andriotis, C., & Papakonstantinou, K. (2019). Managing engineering systems with large state and action spaces through deep reinforcement learning. *Reliability Engineering and System Safety, 191*, 106483.

Arulkumaran, K., Deisenroth, M., Brundage, M., & Bharath, A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine, 34*, 26–38.

Baraldi, P., Bonfanti, G., & Zio, E. (2018). Differential evolution-based multi-objective optimization for the definition of a health indicator for fault diagnostics and prognostics. *Mechanical Systems and Signal Processing, 102*, 382–400.

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Berlin, Heidelberg: Springer-Verlag.

Bousdekis, A., Magoutas, B., Apostolou, D., & Mentzas, G. (2018). Review, analysis and synthesis of prognostic-based decision support methods for condition based maintenance. *Journal of Intelligent Manufacturing, 29*, 1303–1316.

Byon, E., & Ding, Y. (2010). Season-dependent condition-based maintenance for a wind turbine using a partially observed markov decision process. *IEEE Transactions on Power Systems, 25*, 1823–1834.

Chen, Z. (2003). Bayesian filtering: From kalman filters to particle filters, and beyond. *Statistics*, 1–69.

Cheng, F., Qu, L., & Qiao, W. (2018). Fault prognosis and remaining useful life prediction of wind turbine gearboxes using current signal analysis. *IEEE Transactions on Sustainable Energy, 9*, 157–167.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:1412.3555.

Dohi, T., & Okamura, H. (2016). Dynamic software availability model with rejuvenation. *Journal of the Operations Research Society of Japan, 59*, 270–290.

Fang, X., Gebraeel, N. Z., & Paynabar, K. (2017). Scalable prognostic models for large-scale condition monitoring applications. *IISE Transactions, 49*, 698–710.

Gal, Y. (2016). *Uncertainty in deep learning. Oxford machine learning.* Great Britain: Oxford.

Ghavamzadeh, M., Mannor, S., Pineau, J., & Tamar, A. (2015). Bayesian reinforcement learning: A survey. *Foundations and Trends in Machine Learning, 8*, 359–483.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* The MIT Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*, 1735–1780.

Huang, G. B., Zhu, Q. Y., & Siew, C. K. (2006). Extreme learning machine: Theory and applications. *Neurocomputing, 70*, 489–501.

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

Ko, Y. D. (2019). An efficient integration of the genetic algorithm and the reinforcement learning for optimal deployment of the wireless charging electric tram system. *Computers & Industrial Engineering, 128*, 851–860.

Ko, Y. M., & Byon, E. (2017). Condition-based joint maintenance optimization for a large-scale system with homogeneous units. *IISE Transactions, 49*, 493–504.

Kontar, R., Son, J., Zhou, S., Sankavaram, C., Zhang, Y., & Du, X. (2017). Remaining useful life prediction based on the mixed effects model with mixture prior distribution. *IISE Transactions, 49*, 682–697.

Lei, Y., Li, N., Guo, L., Li, N., Yan, T., & Lin, J. (2018). Machinery health prognostics: A systematic review from data acquisition to rul prediction. *Mechanical Systems and Signal Processing, 104*, 799–834.

Liu, Y., Chen, Y., & Jiang, T. (2020). Dynamic selective maintenance optimization for multi-state systems over a finite horizon: A deep reinforcement learning approach. *European Journal of Operational Research, 283*, 166–181.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature, 518*, 529–533.

Olde Keizer, M., Teunter, R., & Veldman, J. (2016). Clustering condition-based maintenance for systems with redundancy and economic dependencies. *European Journal of Operational Research, 251*, 531–540.

Orchard, M., & Vachtsevanos, G. (2009). A particle-filtering approach for on-line fault diagnosis and failure prognosis. *Transactions of the Institute of Measurement and Control, 31*, 221–246.

Park, S., Pil Hwang, J., Kim, E., & Kang, H. J. (2010). Vehicle tracking using a microwave radar for situation awareness. *Control Engineering Practice, 18*, 383–395.

Sahebjamnia, N., Tavakkoli-Moghaddam, R., & Ghorbani, N. (2016). Designing a fuzzy q-learning multi-agent quality control system for a continuous chemical production line–A case study. *Computers & Industrial Engineering, 93*, 215–226.

Saxena, A., & Goebel, K. (2008). Turbofan engine degradation simulation data set. NASA Ames Prognostics Data Repository. https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/. Accessed on March 2020.

Saxena, A., Goebel, K., Simon, D., & Eklund, N. (2008). Damage propagation modeling for aircraft engine run-to-failure simulation.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks, 61*, 85–117.

Shiue, Y. R., Lee, K. C., & Su, C. T. (2018). Real-time scheduling for a smart factory using a reinforcement learning approach. *Computers & Industrial Engineering, 125*, 604–614.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature, 529*, 484–489.

Song, Y., Liu, D., Yang, C., & Peng, Y. (2017). Data-driven hybrid remaining useful life estimation approach for spacecraft lithium-ion battery. *Microelectronics Reliability, 75*, 142–153.

Sun, Q., Ye, Z. S., & Chen, N. (2017). Optimal inspection and replacement policies for multi-unit systems subject to degradation. *IEEE Transactions on Reliability, 67*, 401–413.

Tsoutsanis, E., & Meskin, N. (2017). Derivative-driven window-based regression method for gas turbine performance prognostics. *Energy, 128*, 302–311.

Wang, D., & Tsui, K. L. (2018). Brownian motion with adaptive drift for remaining useful life prediction: Revisited. *Mechanical Systems and Signal Processing, 99*, 691–701.

Wang, X., Wang, H., & Qi, C. (2016). Multi-agent reinforcement learning based maintenance policy for a resource constrained flow line system. *Journal of Intelligent Manufacturing, 27*, 325–333.

Wei, S., Bao, Y., & Li, H. (2020). Optimal policy for structure maintenance: A deep reinforcement learning framework. *Structural Safety, 83*.

Wei, T., Huang, Y., & Chen, C. (2009). Adaptive sensor fault detection and identification using particle filter algorithms. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews, 39*, 201–213.

Xanthopoulos, A., Kiatipis, A., Koulouriotis, D., & Stieger, S. (2017). Reinforcement learning-based and parametric production-maintenance control policies for a deteriorating manufacturing system. *IEEE Access, 6*, 576–588.

Zhao, R., Yan, R., Chen, Z., Mao, K., Wang, P., & Gao, R. (2019). Deep learning and its applications to machine health monitoring. *Mechanical Systems and Signal Processing, 115*, 213–237.

Zhao, X., Gaudoin, O., Doyen, L., & Xie, M. (2019). Optimal inspection and replacement policy based on experimental degradation data with covariates. *IISE Transactions, 51*, 322–336.

Zhou, X., Xi, L., & Lee, J. (2009). Opportunistic preventive maintenance scheduling for a multi-unit series system based on dynamic programming. *International Journal of Production Economics, 118*, 361–366.