Computers and Chemical Engineering 165 (2022) 107956

Contents lists available at ScienceDirect

Computers and Chemical Engineering

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

Review

A tutorial review of neural network modeling approaches for model predictive control

Yi Ming Ren [a], Mohammed S. Alhajeri [a,c], Junwei Luo [a], Scarlett Chen [a], Fahim Abdullah [a], Zhe Wu [d], Panagiotis D. Christofides [a,b,*]

[a] Department of Chemical and Biomolecular Engineering, University of California, Los Angeles, CA, 90095-1592, USA
[b] Department of Electrical and Computer Engineering, University of California, Los Angeles, CA 90095-1592, USA
[c] Department of Chemical Engineering, Kuwait University, Safat 13060, Kuwait
[d] Department of Chemical and Biomolecular Engineering, National University of Singapore, 117585 Singapore, Singapore

ARTICLE INFO

Keywords:
Time-series forecasting
Feed-forward neural networks
Recurrent neural networks
Encoder–decoder architecture
Model predictive control
Nonlinear systems
Chemical processes

ABSTRACT

An overview of the recent developments of time-series neural network modeling is presented along with its use in model predictive control (MPC). A tutorial on the construction of a neural network-based model is provided and key practical implementation issues are discussed. A nonlinear process example is introduced to demonstrate the application of different neural network-based modeling approaches and evaluate their performance in terms of closed-loop stability and prediction accuracy. Finally, the paper concludes with a brief discussion of future research directions on neural network modeling and its integration with MPC.

1. Introduction

Model predictive control (MPC) has attracted significant research interest as it is one of the major achievements in the development of advanced multivariate process control systems due to its ability to compute the optimal control actions based on not only the instantaneous state measurements but also the anticipated process response. Specifically, MPC relies on its built-in linear/nonlinear model to capture the dynamic behavior of the process and predict the response of the process over a finite horizon window, such that it can determine the optimal control trajectory by solving a dynamic optimization problem subject to input and state constraints at every sampling time. Attributed to the achievement of developing self-tuning controllers in the 1970s, such as minimum variance (MV), generalized minimum variance (GMV) (Clarke and Gawthrop, 1975, 1979), and Pole Placement (Wellstead et al., 1979) controls, the methodology of MPC was proposed. Since its conceptualization, MPC has been developed and modified in various ways over the past few decades. Rawlings (2000) provided a tutorial review of MPC approaches for control practitioners and the variables/parameters that must be considered when designing an MPC. Mayne et al. (2000) carried out an exhaustive literature review of the advances in MPC to handle hard constraints in both linear and nonlinear systems, with an emphasis on the results regarding stability and optimality. Morari and Lee (1999) summarized the previous 15 years of research in MPC and proposed several new directions for the future such as performance monitoring, process diagnostics, estimating states in nonlinear systems, improving system identification in the multi-input multi-output (MIMO) system context, and the challenges and reliability of the on-line dynamic optimization problem. Following these advances, MPC has been widely accepted in industry (Holkar and Waghmare, 2010). For example, in Hrovat et al. (2012), MPC was implemented in an industrial automotive system and demonstrated superior closed-loop performance compared to traditional control schemes. However, as stated in Mohanty (2009), Amrit et al. (2011), Ellis et al. (2014), Angeli et al. (2011), a reliable process model that can capture the input–output relation of the dynamic system is essential to the success of advanced model-based control systems (e.g., Lyapunov-based MPC (LMPC) and economic MPC (EMPC)) as these systems require process models with well-characterized accuracy to predict the temporal evolution of the states, and thus, the identification of process models is a central pillar of control science and engineering.

Traditionally, mathematical and statistical models are widely used to derive process models in the field of chemical engineering. For instance, famous first-principles models such as the Navier–Stokes equations are crucial to modeling the fluid dynamics within physical systems. However, deriving first-principles models for chemical processes

* Corresponding author at: Department of Chemical and Biomolecular Engineering, University of California, Los Angeles, CA, 90095-1592, USA.
E-mail address: pdc@seas.ucla.edu (P.D. Christofides).

https://doi.org/10.1016/j.compchemeng.2022.107956
Received 24 May 2022; Received in revised form 30 July 2022; Accepted 9 August 2022
Available online 13 August 2022
0098-1354/© 2022 Elsevier Ltd. All rights reserved.

can be challenging due to the complexity of determining the fundamental physico-chemical phenomena of a process. In contrast, modeling dynamic systems using data-driven models has attracted significant attention, both historically and recently. In the context of developing dynamic models to embed into controllers, linear models have been studied exhaustively over the past few decades, leading to a well-developed framework and literature in this field. This is primarily due to the mathematical simplicity and results that can be derived for controllers with linear process models, despite the fact that most chemical processes exhibit highly nonlinear behavior and are characterized by numerous complex interactions between variables as seen in distillation columns (Lévine and Rouchon, 1991) and catalytic continuous stirred-tank reactors (CSTR) (e.g., Chang and Aluko, 1984). For industrial process control systems, the parameters of a linear data-driven model are typically identified from industrial or simulation data (e.g., Wilson and Sahinidis, 2017). A category of such data-driven models is autoregressive models such as autoregressive with exogenous inputs (ARX) or autoregressive-moving-average model with exogenous inputs (ARMAX) (e.g., Maner and Doyle III, 1997). An ARX model takes the weighted sum of the lagged states and inputs to predict the current states and may also be reformulated as a linear time-invariant state–space model, which has been studied in-depth in the control literature. The primary reason linear models are still employed in the control of nonlinear processes is that, besides the aforementioned mathematical considerations, they are often able to be used for the design of a controller that can stabilize the nonlinear process itself. For example, Alanqar et al. (2015b) investigated the design of a Lyapunov-based EMPC (LEMPC) using empirical linear state–space models, derived conditions to guarantee closed-loop stability of the nonlinear process under the LEMPC based on the linear empirical model, applied the controller to a nonlinear chemical process, and found it to be a computationally efficient framework. Furthermore, when using a linear state–space model as the process model in MPC, its structure becomes similar to that of a linear–quadratic regulator (LQR), which has convex properties and guaranteed convergence (Chow et al., 1975). In practice, this can lead to faster convergence of the MPC optimization problem compared to the usage of nonlinear models in MPC.

While the above methods perform well for linear systems, process modeling using linear models continues to be challenging for large-scale complex processes due to the limitation of enough and flexible parameters to capture all nonlinearities in the system. Even though linearized models can still provide accurate approximations around steady-state regions, as the process deviates from the steady-state regions, linear models' performance suffers because the linearization approximation no longer holds. In response, nonlinear autoregressive models such as nonlinear autoregressive with exogenous inputs (NARX) are constructed where a more flexible nonlinear mapping can be added to the lagged states and inputs to capture the nonlinear coupling between known input effects and unknown input effects (Billings, 1980; Pemberton, 1990; Henson and Seborg, 1997). One problem that autoregressive models have is the large number of parameters that need to be estimated. Therefore, extensive research has been done regarding methods to reduce the dimension of these models. For example, by using input projection methods such as principal component analysis (PCA) and partial least squares (PLS), the dimensions of autoregressive models can be reduced to a reasonable amount (Qin and McAvoy, 1992). In addition, constraints to the number of parameters can be added to the model such as in the case of nonlinear additive auto regressive model with exogenous input (NAARX) (Henson and Seborg, 1997). However, even with the use of dimension reduction algorithms, it is still a laborious task, especially for multivariable systems, to balance between choosing a suitable model structure and identifying all the required parameters (Lee, 2000). The field of nonlinear dynamic modeling remains an active area of research. In the recent literature, several methods have been proposed including explanatory approaches such as sparse identification for nonlinear dynamical systems (SINDy) (Brunton et al., 2016), which directly identifies a nonlinear system as a first-order ordinary differential equation that may be integrated in time. For systems such as chemical processes that evolve on an attractor and reach a steady-state, explanatory methods such as SINDy have been shown to accurately capture the long-term trajectories of nonlinear systems in the presence of time-scale multiplicities (Abdullah et al., 2021a) or high levels of sensor noise in the data (Abdullah et al., 2022). The resulting SINDy models have also demonstrated closed-loop stability and faster convergence than first-principles models when incorporated into an MPC (Abdullah et al., 2021b). In Alanqar et al. (2015a), well-conditioned polynomial nonlinear state–space models were developed to be incorporated into an LEMPC. The model identification explicitly accounted for conditioning to yield well-conditioned models that could be integrated with a relatively large integration time-step, leading to significant reduction in computation time per sampling period. Other nonlinear dynamic modeling methods in the literature opt for data-driven approaches such as Runge–Kutta time-steppers embedding neural networks to handle nonlinearities (González-García et al., 1998; Fablet et al., 2018; Raissi et al., 2018; Rudy et al., 2019) and entropic regression (AlMomani et al., 2020). It is noted that these alternative methods exist and have their advantages as well as disadvantages, but a comparison study is beyond the scope of this paper.

Entering the era of big data, deep learning (DL) methods such as artificial neural networks (ANN) have gained much attention for their exceptional performance in capturing the behavior of complex physical systems, making them top candidates for model-based control systems. Deep learning is a subcategory of machine learning that focuses on the use of neural networks for different tasks. The large number of tunable parameters and the rich variety of nonlinear functions ANNs possess allow the capture of previously-considered "difficult nonlinearities". When given the appropriate parameters, the universal approximation theorem states that an ANN is capable of capturing any complex input–output relation (Csáji et al., 2001; Lu et al., 2019). However, ANNs did not gain popularity until recently due to the difficulties associated with implementation and training (Qin and McAvoy, 1992). Specifically, for the modeling of complex physical systems, especially noisy and/or nonlinear processes, *deep* ANNs, or Deep Neural Networks (DNN), which are ANNs with a large number of intermediate layers, may be necessary to capture the input–output relation. Compared to traditional machine learning algorithms, DNNs reduce the need for extensive feature engineering and can learn features directly from input data (LeCun et al., 2015). Although training DNNs requires a large volume of data due to the high number of parameters to be estimated, which renders their implementation generally difficult, several recent developments have mitigated the challenges of training DNNs. Firstly, the proliferation of data generated by machines and devices in the last decade has made the training of data-demanding DNNs viable (Yin and Kaynak, 2015). Secondly, while DNNs inherently involve many matrix operations, leading to large demands for computational power during training, recent advancements in computing infrastructure including cloud and parallel computing have made training DNNs feasible. Finally, the development of open-source machine learning libraries such as Tensorfow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and Keras have made the construction and training of complex ANNs much more straightforward and accessible.

There are many different types of ANNs and DNNs; specifically, feed-forward neural networks (FNN) and recurrent neural networks (RNN) and their variants (e.g., Chow and Fang, 1998; Schmidhuber, 2015; Gurney, 2018) have demonstrated potential for use in model-based control systems. Throughout the scientific history, FNNs have been proposed to be used as process models in MPC. Draeger et al. (1995) proposed the use of a two-hidden-layer FNN as the process model for dynamic matrix control (DMC), which is an early linear form of MPC. A more recent use of FNN is to model nonlinear dynamic processes since the multivariate nature of these processes leads

to internal state interactions, causing difficulties for traditional models (Afram et al., 2017). Therefore, research has been conducted to exploit the nonlinear nature of FNNs to develop process models for MPC. In Mohanty (2009), a dynamic FNN-based MPC was designed to control the interface level of a flotation column by manipulating the tailings flow rate. The dynamic FNN took in a multi-time step history of the states to predict the interface level after one-time step. The MPC's prediction horizon is set to contain multiple sampling periods, and each sampling period is equal to the prediction length of the dynamic FNN. Results showed that the dynamic FNN-based MPC performed better than traditional PID controllers due to faster convergence and smoother trajectories. Kittisupakorn et al. (2009) reported the use of a stacked FNN-based MPC for a multivariable nonlinear steel pickling process. FNNs were stacked in series, and an iterative method was used to obtain different future time steps of process states within the MPC's prediction horizon. The simulation results show that the FNN-based MPC displayed good convergence and stability even under disturbances and noise. In addition to using FNNs as the process models, FNNs can also be used as a model identifier to determine when a process is subject to parameter variations and uncertainties (Hedjar, 2013). The identifier FNN updates the weights of the process model, which is a separate predictor FNN, in case of system variations and thus creates an adaptive neural network-based MPC. While the above use of FNNs as process models in MPC displayed good results, modeling complex long-term dynamic processes was not intuitive given the unidirectional structure of FNNs.

Therefore, the idea of incorporating recurrency to neural networks was proposed to better capture the ordinal nature within time-series datasets. The first RNNs can be traced back to the 1980s, when Hopfield networks were first created for pattern recognition purposes (Hopfield, 1982; Rumelhart et al., 1986). There were also attempts to use RNNs to model nonlinear dynamic processes, but they were not widespread due to difficulties to train generalizable and accurate RNNs (Miller et al., 1995). However, with recent progress in technology and neural network structure, RNNs have now emerged as a leading method to model nonlinear dynamic processes (Schuster and Paliwal, 1997; Schmidhuber, 2015). Esche et al. (2022) evaluated the applicability of several neural network architectures, including FNN and RNN variants, for different dynamic processes as surrogate models. For dynamic processes, FNNs may not perform as well as RNNs due to the lack of feedback connections that introduce past information derived from earlier inputs into the current output. Furthermore, the incorporation of feedback loops in RNNs leads to capturing dynamic behavior in a way conceptually similar to nonlinear dynamic models derived from first-principles (Miljanovic, 2012). As a result, modern RNN-based process models (e.g., Long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997), Gated recurrent unit (GRU) (Cho et al., 2014a), and encoder–decoder (Sutskever et al., 2014)) have been integrated with different model-based control schemes and used in many research fields, including chemical (Zheng et al., 2022), mechanical (Xu et al., 2016), and pharmaceutical (Wong et al., 2018) engineering. Specifically for process control, Zarzycki and Ławryńczuk (2021) has conducted a comparative study on the use of LSTM- and GRUs as dynamic process models in predictive control for two chemical reactors. It was found that both models approximated the properties of the dynamic systems with high accuracy and drove the systems to desired set-points. For certain stochastic processes, a single RNN process model may not be adequate to capture the complex nature of the system; therefore, multiple RNNs can be used in conjunction, creating an ensemble in which the mean or median of the ensemble is taken as the final prediction. Wu et al. (2019b) utilized an ensemble of RNNs as the MPC process model to model the behavior of a fixed-bed catalytic reactor and demonstrated its performance with respect to computational fluid dynamics (CFD) results. In addition to traditional RNN architectures, an encoder–decoder architecture was proposed by Cho et al. (2014b) in which two RNNs were used in series to capture input sequences of various lengths and long-term dependencies. Zhang et al. (2021b) have demonstrated that, for dynamic processes that contain many long-term dependencies, encoder–decoder-based RNNs perform better than LSTM/GRU-based RNNs. Li and Tong (2021) have adopted an encoder–decoder RNN model to develop an MPC for the control of an HVAC system and demonstrated good convergence and stability. In addition to performance, Ellis and Chinde (2020) have argued that encoder–decoder models can easily be constructed from a model definition perspective for HVAC systems. Specifically, the list of inputs and outputs of encoder–decoder model is clearly aligned with the inputs and outputs of the HVAC process and this may help simplify model construction and reduce training costs of EMPC. Finally, Bonassi et al. (2022) discusses the incorporation and evaluation of different RNN structures in MPC. Specifically, these RNNs were assessed from the perspective of a control system designer with emphasis on stability guarantees, safety verification, and consistency with the physical system for the RNN models. Wu et al. (2020) proposed that the incorporation of physical knowledge into RNN models improves the performance of the MPC system. This claim was further supported by Alhajeri et al. (2022), in which two RNN models, one with physical knowledge and one without, were compared against each other for a two-CSTR-in-series system simulated on a high-fidelity chemical process simulator. It was found that the physics-based RNN-MPC system converged faster and required less computational time.

This article aims to survey the popular neural network modeling approaches and provide a tutorial on the construction and integration of these models with MPC. Recent advances in the development of neural network models for specific scenarios such as noisy data and on-line adaptation learning are presented as remarks throughout the tutorial. The explanations presented in this paper are meant to be accessible to a beginning graduate student with limited knowledge in control and machine learning. The remainder of the paper is organized as follows: in the next section, preliminary knowledge on the class of systems considered and stability assumptions are presented. The third section discusses the concept of MPC and real-time optimization (RTO) and their implementation in process control. In addition, the role of process model within MPC and RTO is presented with a focus on neural network model-based MPC and RTO. In the fourth section, the theories behind neural networks are discussed with an emphasis on the intuition behind their architecture. This survey on neural network models is not meant to be comprehensive, but to give readers an idea on the evolution of neural networks and background theories. The fifth section aims to give a brief tutorial on the construction of a neural network model. Starting from problem identification to model evaluation, the workflow is meant to be iterative and change with new findings during application. The sixth section gives an example of applying different neural network model-based MPCs and their performance on a chemical process. Finally, a summary and future directions of neural networks for MPC are discussed.

2. Preliminaries

2.1. Notation

The notation $|\cdot|_1$ is used to represent the L_1 norm, and $|\cdot|$ is used to represent the Euclidean norm. A function $f(\cdot)$ is of class C^1 if it is continuously differentiable in its domain. $\odot$ denotes the Hadamard product or element-wise multiplication. $:=$ denotes the assignment operator. The expression $x \rightarrow c$ represents the variable x approaching some constant c.

For time step notation, each time step represents a single integration step h_c, and Δ represents the sampling time, which is the time interval at which state measurements are available. $t = t_k$ is defined as the current time step, and any time before $t = t_k$ is considered historical information. M is the number of inputs that are fed to the model for prediction, which can include historical and present values of the state

and manipulated input variables. N is the number of outputs that are produced by the model, which indicates the predicted process states in the next N time steps with an interval of h_c. The predicted outputs from the machine learning model constitute the intermediate and final states within and at the end of one sampling period Δ, respectively, where $\Delta = N \cdot h_c$. For each prediction, a sequence of historical and present information $(t_{k-M+1}, \ldots, t_k)$ is used to predict the future trajectory over a single sampling period $(t_{k+1}, \ldots, t_{k+N})$. Since the machine-learning model provides N future predictions over one Δ, the model will be called iteratively K times to solve the MPC with a prediction horizon of K sampling periods.

Finally, for variable notation in this paper, x is used to denote two objects depending on the context. In the context of control, x denotes the states within the system. In the context of machine learning, x denotes the input to the model. This is to keep the notation consistent with other work in the fields of both control systems and machine learning.

2.2. Class of systems

In this work, the class of multi-input multi-output nonlinear continuous-time systems is considered and can be represented by the following state–space form:

$$\dot{x} = F(x,u) = f(x) + g(x)u \tag{1a}$$

$$y = h(x) \tag{1b}$$

where $x \in \mathbf{R}^a$ denotes the state vector, $u \in \mathbf{R}^b$ is the vector of manipulated inputs and $y \in \mathbf{R}^a$ represents the vector of state measurements which are available at every sampling period. The input vector is bounded by $u \in U = \{u_i^{\min} \leq u_i \leq u_i^{\max}, i = 1, \ldots, m\} \subset \mathbf{R}^b$. The terms $f(\cdot)$, $g(\cdot)$, and $h(\cdot)$ are sufficiently smooth vector and matrix functions of dimensions $n\times 1$, $n\times m$, and $n\times 1$, respectively, with the assumption that state and input variables are in deviation from their steady-state values such that the origin is a steady-state of the nominal system of Eq. (1) (i.e., $(x_{ss}^*, u_{ss}^*) = (0,0)$, where the subscript "$ss$" indicates the steady-state). Throughout this tutorial review, initial time is assumed to be $t_0 = 0$, and all states are assumed to be measurable.

2.3. Stabilizability assumption

The existence of a stabilizing feedback control law of the form $u = \Phi(x) \in U$ is assumed for stability considerations. The objective of this controller is to ensure that the origin of the nominal system of Eq. (1) is exponentially stable under this controller. This assumption implies that there is a control Lyapunov function of class C^1 denoted as $V(x)$ such that the following inequalities hold for all x within an open neighborhood D around the origin:

$$c_1|x|^2 \leq V(x) \leq c_2|x|^2, \tag{2a}$$

$$\frac{\partial V(x)}{\partial x} F(x, \Phi(x)) \leq -c_3|x|^2, \tag{2b}$$

$$\left|\frac{\partial V(x)}{\partial x}\right| \leq c_4|x| \tag{2c}$$

where c_1, c_2, c_3, c_4 are all positive real numbers. The controller $\Phi(x)$ can be constructed in the form of the universal Sontag control law proposed in Lin and Sontag (1991). Following Wu et al. (2019c), first, the open neighborhood around the origin, D, is identified where Eq. (2b) holds under the stabilizing controller $u = \Phi(x) \in U$. Then, the closed-loop stability region Ω_ρ is identified as a level set of $V(x)$ within the region D where $\Omega_\rho = \{x \in D \mid V(x) \leq \rho\}$, with $\rho > 0$. Moreover, the Lipschitz property of $F(x,u)$ with the upper and lower bounds on u implies the existence of positive constants Γ, L_x, L_x' such that the following inequalities hold for all x and $x' \in D$, and $u \in U$:

$$|F(x,u)| \leq \Gamma \tag{3a}$$

$$|F(x,u) - F(x',u)| \leq L_x|x - x'| \tag{3b}$$

$$\left|\frac{\partial V(x)}{\partial x} F(x,u) - \frac{\partial V(x')}{\partial x} F(x',u)\right| \leq L_x'|x - x'| \tag{3c}$$

3. Real-time optimization (RTO) and model predictive control (MPC)

3.1. Real-time optimization (RTO)

Real-time optimization (RTO) system is an advanced model-based process optimization and control tool to compute the optimum operating steady-state condition based on a user-defined objective function (e.g., minimum energy consumption, maximum economic profitability, etc. (Zhang et al., 2019)). Typically, RTO is computed over a significantly longer window than the supervisory control layer. For example, RTO may be computed over hours or days while the supervisory control layer may be computed over minutes. As a result, the setpoint is updated in real-time according to the RTO output by addressing the optimization problem described in Eq. (4):

$$\begin{aligned} \min_{x,u} \quad & L_e(x,u) \\ \text{s.t.} \quad & F(x,u) = 0 \\ & g_p(x,u) \leq 0 \\ & g_e(x,u) \leq 0 \end{aligned} \tag{4}$$

which minimizes a user-defined cost function expressed by $L_e(x,u)$. This function is commonly termed the economic cost function or economic stage cost, since it is a direct or indirect reflection of the process economics. The objective function can be designed to maximize typical chemical engineering performance measures such as the production rate of the desired product, selectivity of the desired product, and product yield. F is a steady-state model. As for g_p, it is a vector function, and represents process constraints (i.e., physical constrains) such as limits on inputs, and g_e is the economic constraints that includes oil and energy prices, etc. Both terms are matrix functions of dimensions $\mathbf{R}^p$ and $\mathbf{R}^e$, respectively.

3.2. Model predictive control (MPC)

MPC is an advanced control scheme that is able to anticipate the future states of the process to make intelligent decisions with respect to the constraints of the process. Theoretically, MPC consists of three main components (i.e., an objective function, a process model, and a real-time optimizer (Camacho and Bordons, 2013)), where the process model provides the prediction of state responses based on the underlying physical and chemical phenomena of the system. Subsequently, the real-time optimizer will compute the optimal control actions $u^*(t)$ for each sampling period (denoted by Δ) within the prediction horizon by solving a finite-horizon optimization problem with respect to the objective function and constraints. The operation of MPC is to address the following optimization problem:

$$\min_{u \in S(\Delta)} \int_{t_k}^{t_k+K\Delta} L(\hat{x}(\zeta), u(\zeta))\, d\zeta \tag{5a}$$

$$\text{s.t.} \quad \dot{\hat{x}} = F(\hat{x}(t), u(t)) \tag{5b}$$

$$\hat{x}(t_k) = x(t_k) \tag{5c}$$

$$u(t) \in U, \forall t \in [t_k, t_k + K\Delta) \tag{5d}$$

$$g(\hat{x}(t), u(t)) \in G, \forall t \in [t_k, t_k + K\Delta) \tag{5e}$$

where $\hat{x}$ represents the predicted states of the process model and $S(\Delta)$ denotes the set of piecewise constant functions with Δ. The optimal control actions are calculated to minimize the time integral of the objective function $L(\hat{x}(t), u(t))$ in Eq. (5a) over the prediction horizon K, meaning

that the predicted trajectory over $K\Delta$ into the future is accounted for within the optimization problem. However, only the control action for the first sampling period will be implemented to the process system. In other words, K optimal input actions are computed for each Δ from the current time step $t = t_k$ until $t = t_k + K\Delta$ in a feedback control manner on the basis of the predicted state trajectory $\hat{x}$, but only the first optimal input $u^*(t_k)$ is applied to the process over the next sampling period and is held constant over Δ, which is known as sample-and-hold implementation. The predicted state trajectory $\hat{x}$ is computed using the process model of Eq. (5b), which can be a first-principles or a data-driven model. Moreover, as shown in Eq. (5c), the initial condition for the process model is obtained from the real-time measurement. In addition, the feasible range of control actions is defined in Eq. (5d). Lastly, Eq. (5e) represents any additional constraints (e.g., Lyapunov stability constraints (Wu et al., 2019c)) that are imposed to ensure that the optimal control actions meet the necessary conditions to guarantee closed-loop stability under sample-and-hold implementation.

Remark 1. The MPC is implemented in a receding horizon manner so that, at every sampling time, the optimization problem is solved again when new feedback measurements are received. Note that state and input variables are typically represented in their deviation forms such that the steady-state values are at the origin. The operating steady-state values can be the optimal states and inputs that are calculated by the RTO, which is executed at a slower frequency compared to MPC.

Remark 2. The mathematical setup of MPC and RTO is similar to each other, but RTO is based on steady-state process models and MPC is based on dynamic models (Qin and Badgwell, 2003). Therefore, both functionalities are critically dependent on the respective process models. As mentioned, the first-principles models are the primary candidates for this role. However, in practical operations, oftentimes an accurate first-principles model is not available. Thus, by replacing the process model with a neural network model, machine learning can be integrated with RTO and MPC to accomplish their respective goals using a data-driven modeling approach.

3.3. Multi-layer control scheme

Traditional architecture-process optimization, particularly with regard to economic considerations, and chemical process control have been resolved in a hierarchical multi-layer control scheme, as illustrated in Fig. 1. Specifically, the first layer is the RTO, which solves for optimal set points in accordance with the steady-state process model and supervises the sublayers by sending out the optimization results. MPC is implemented in the second layer of this control scheme to direct and derive the process to the new operating point through manipulating the inputs with its constrained optimal control methodology, which takes into account physical limitations, process variable interactions, and predicted responses. Control actions from the MPC are employed to the system by the regulatory control layer.

In addition to the top two-layer control scheme, an EMPC, which introduces penalization terms into the objective function based on the economic performance of the process, has been proposed for its ability to achieve similar functionality without RTO and has attracted increasing attention recently (Amrit et al., 2011; Ellis et al., 2014). However, regardless of the control scheme, a reliable process model is one of the most critical components of MPC and EMPC.

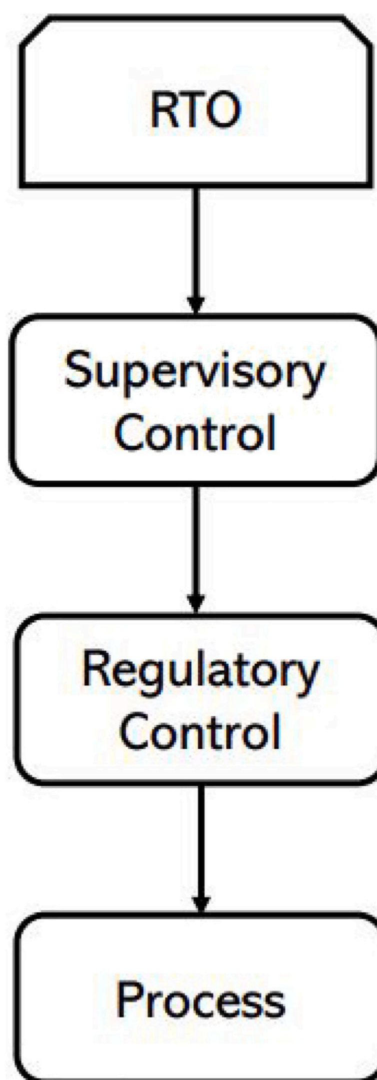


Fig. 1. The typical process optimization and control perspective used in the field of chemical process industries.

4. Neural network architecture overview

4.1. Feed-forward neural networks (FNN)

Feed-forward neural networks are generally made up of multiple layers of neurons, with each neuron being a single logistic unit. Figs. 2(a) and (b) show the schematics of an FNN and the model graph describing the calculations conducted within a logistic unit, respectively. The main concept behind the FNN or any neural network can be summarized into two parts: a forward pass to predict the desired output and a backward pass to update the weights and biases. In the forward pass, the input features are propagated through the neural network to generate the predicted output. In the backward pass, the predicted output is compared with the true output, and the error is back propagated through all the logistic units. The weights and biases of each logistic unit are updated with respect to the error to improve the next prediction.

Similar to how real neurons operate in the human brain, the forward propagation of a logistic unit functions analogously and is described mathematically by the following equations:

$$z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} h_j^{[l-1]} w_{jk}^{[l]} + b_k^{[l]} \tag{6a}$$

$$h_k^{[l]} = g^{[l]}(z_k^{[l]}) \tag{6b}$$

where superscripts in brackets, $[\cdot]$, denote the layer number of the neural network. l denotes the lth layer of the neural network that varies between 0 and L. Only generalized intermediate layer equations within a neural network are shown in Eq. (6). The input layer is represented

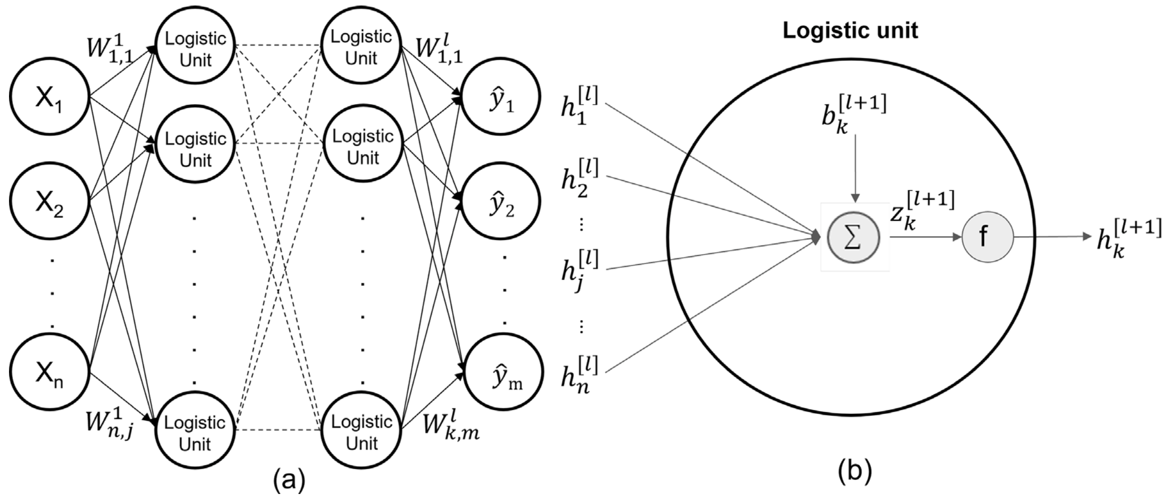


Fig. 2. Schematics of (a) general FNN structure and (b) logistic unit within FNNs.

by $l = 0$ where $h_j^{[0]}$ is the equivalent of the input x_j and the output layer is represented by $l = L$ where $h_j^{[L]}$ is the equivalent of the predicted output $\hat{y}_j$. Subscripts j and k denote the jth and kth units within their respective layer in the neural network, b denotes the bias, and h denotes the hidden state. $w_{jk}^{[l]}$ is the weight connecting the jth neuron of the layer $l-1$ to the current kth neuron in the layer l. z denotes the weighted hidden states and the bias term. $n^{[l]}$ is the total number of neurons in the lth layer. In the forward propagation step, first, following Eq. (6a), each logistic unit takes inputs from all connected units from the previous layer, $h_j^{[l-1]}$, and aggregates the input signals together with its own bias, $b_k^{[l]}$. The summed signal, $z_k^{[l]}$, then undergoes a certain activation function, $g^{[l]}$, with common ones being the rectified linear unit (ReLu) or *tanh*, before being sent as the input $h_k^{[l]}$ to the next layer as per Eq. (6b). During training and inference, signals start from the input layer and propagate to the final output layer L using this two-step procedure.

$$E = \frac{1}{R}\sum_{i=1}^{R} L(\hat{y}, y) \tag{7}$$

where E is the cost function, L is the loss function for each sample, i is the index of data samples, R is the total number of data samples, $\hat{y}$ is the predicted output, and y is the true output. During training, the cost function is minimized using optimization algorithms to find the optimal set of weights and biases.

Optimization algorithms are used to update the weights and biases within the FNN. Gradient descent (GD), when applied to weights and biases, respectively, is shown in Eq. (8) below, and its variants are commonly used as the optimization algorithm for neural network training. The general structure of the GD algorithm is of the form,

$$w_{jk}^{[l]} := w_{jk}^{[l]} - \alpha \frac{\partial E}{\partial w_{jk}^{[l]}} \tag{8a}$$

$$b_k^{[l]} := b_k^{[l]} - \alpha \frac{\partial E}{\partial b_k^{[l]}} \tag{8b}$$

where $w_{jk}^{[l]}$ and $b_k^{[l]}$ are being updated and α is the learning rate and is conventionally positive (i.e., $\alpha > 0$). The main idea behind gradient descent is to find the local minimum of a differentiable function by iteratively moving in the opposite direction of the gradient of the function at each point. During neural network training, the gradient descent algorithm minimizes the loss at each iteration or epoch by varying the weights and biases. In machine learning, a single epoch refers to one complete pass of the training dataset through the algorithm. As shown in Eq. (8), the gradient of the cost function with respect to each weight and bias needs to be calculated at each layer using the backpropagation algorithm. The backpropagation algorithm calculates the gradients starting from the final output layer and propagates to the input layer, which ensures computational efficiency by avoiding redundant calculations of intermediate terms (Goodfellow et al., 2016). Using the chain rule, the gradient can be transformed into two partial derivatives involving $z_k^{[l]}$ as shown below:

$$\frac{\partial E}{\partial w_{jk}^{[l]}} = \frac{1}{R}\sum_{i=1}^{R} \frac{\partial L_i}{\partial w_{jk}^{[l]}} = \frac{1}{R}\sum_{i=1}^{R}\left(\frac{\partial L_i}{\partial z_k^{[l]}}\frac{\partial z_k^{[l]}}{\partial w_{jk}^{[l]}}\right) \tag{9a}$$

$$\frac{\partial E}{\partial b_k^{[l]}} = \frac{1}{R}\sum_{i=1}^{R} \frac{\partial L_i}{\partial b_{jk}^{[l]}} = \frac{1}{R}\sum_{i=1}^{R}\left(\frac{\partial L_i}{\partial z_k^{[l]}}\frac{\partial z_k^{[l]}}{\partial b_k^{[l]}}\right) \tag{9b}$$

where L_i is the loss for the ith sample. For weights, in Eq. (9a), the partial derivative of $z_k^{[l]}$ with respect to $w_{jk}^{[l]}$ is the incoming hidden state of the previous layer, $z_j^{[l-1]}$. For biases, the partial derivative of $z_k^{[l]}$ with respect to $b_k^{[l]}$ is simply 1 due to the bias being a constant. Finally, as shown in Eq. (9), the partial derivative of L_i with respect to $z_k^{[l]}$ is also needed to calculate the partial derivative of L_i with respect to w and b. This partial derivative is often denoted as the error term, $\delta_k^{[l]}$. Using the aforementioned simplifications, the gradient of the loss function with respect to weights and biases can be transformed into the following equations:

$$\delta_k^{[l]} = \frac{\partial L_i}{\partial z_k^{[l]}} \tag{10a}$$

$$\frac{\partial L_i}{\partial w_{jk}^{[l]}} = \delta_k^{[l]} z_j^{[l-1]} \tag{10b}$$

$$\frac{\partial L_i}{\partial b_k^{[l]}} = \delta_k^{[l]} \tag{10c}$$

where $\delta_k^{[l]}$ is the error of the kth neuron within the lth layer. The mathematical formulation of the error term is shown by the following equations:

$$\delta_k^{[L]} = (\hat{y} - y) g'(z_k^{[L]}) \tag{11a}$$

$$\delta_k^{[l]} = g'(z_k^{[l]})\left(\sum_{r=1}^{n^{[l+1]}} \delta_r^{[l+1]} w_{kr}^{[l+1]}\right) \tag{11b}$$

where $g'(\cdot)$ is the derivative of the activation function $g(\cdot)$. $\delta_k^{[l]}$, as defined in Eq. (10a), is calculated starting from the final output layer and propagated through to the input layer. At the output layer, assuming square loss, $\delta_k^{[L]}$ is equal to the difference between the true and predicted output multiplied by the derivative of the activation function,

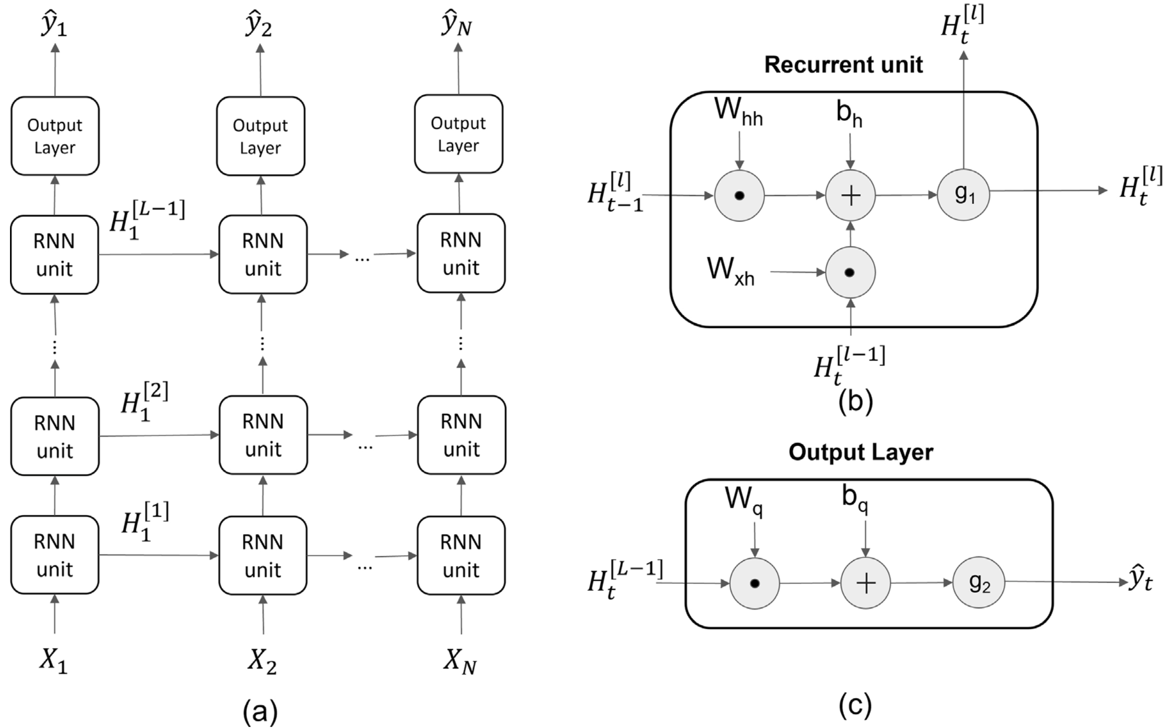


Fig. 3. Schematics of (a) general unfolded RNN structure, (b) recurrent unit, and (c) output layer within RNNs.

$g'(z_k^{[L]})$, as shown in Eq. (11a). For all other layers, the calculation of $\delta_k^{[l]}$ requires the errors from the next layer $\delta_k^{[l+1]}$ and the derivative of the current activation function, $g'(z_k^{[l]})$ as shown in Eq. (11b).

In this section, all equations are shown in vector form with individual neurons and weight connections labeled with subscripts j and k. In the following sections, all equations will be shown in their capitalized matrix form for simplicity. For example, Eqs. (6a) and (6b) are rewritten in their matrix form as follows:

$$Z^{[l]} = H^{[l-1]}W^{[l]} + b^{[l]} \tag{12a}$$

$$H^{[l]} = f^{[l]}(Z^{[l]}) \tag{12b}$$

where the dimensions of each variable in Eq. (12) are as follows: $H^{[l-1]} \in \mathbf{R}^{n\times d}, H^{[l]} \in \mathbf{R}^{n\times h}, Z^{[l]} \in \mathbf{R}^{n\times h}, W^{[l]} \in \mathbf{R}^{d\times h}$, and $b^{[l]} \in \mathbf{R}^{1\times h}$ where the superscripts of $\mathbf{R}$ represent the dimensions of the matrix. Specifically, n is the batch size, which is the number of samples that will be propagated through the neural network and ranges from 1 to N, d is the input size, and h is the number of hidden units. Data samples are fed into the model in batches to promote computational efficiency during training.

4.2. Sequential neural network models

Sequential data is prevalent in many real-world problems and machine learning tasks—most popularly known for its role in natural language processing (NLP) and signal processing. Specifically, in chemical engineering, sequential data can exist in the form of sensor measurements. Sequential models are designed to account for the ordinal nature of these datasets. The main focus of this work will be on neural network model structures that are widely used to conduct time-series forecasting tasks for MPC.

4.2.1. Recurrent neural network (RNN)

RNNs can be thought of as FNNs with two dimensions instead of one, as shown by the unfolded diagram of an RNN in Fig. 3(a). FNN inputs are batches of feature vectors $X_{FNN} \in \mathbf{R}^{n\times d}$, while RNN inputs are batches of sequential feature vectors $X_{RNN} \in \mathbf{R}^{n\times d\times t}$. Therefore, an additional dimension, t, is added to the neural network to account for the data's ordinal property. In FNNs, the output of each neuron, h, is propagated through the network to the last layer before making a final prediction. In multilayer RNNs, h is passed to both the next layer and the next ordinal input. H is a matrix that contains all the hidden states, h, within the same layer. Most RNNs are made up of two different types of units: a recurrent unit and an output unit. The recurrent unit inside the RNN takes in two inputs: the current input vector, X_t, and the previous hidden state, H_{t-1}. For simplicity, all following equations will omit the superscript l that refers to the layer number in the case of multilayer RNNs.

Forward propagation within RNNs is similar to that within FNNs and is described by the following equations:

$$H_t = g_1(X_t W_{xh} + H_{t-1}W_{hh} + b_h) \tag{13a}$$

$$O_t = g_2(H_t W_{hq} + b_q) \tag{13b}$$

where $X_t \in \mathbf{R}^{n\times d}, H_t, H_{t-1} \in \mathbf{R}^{n\times h}, W_{xh} \in \mathbf{R}^{d\times h}, W_{hh} \in \mathbf{R}^{h\times h}, b_h \in \mathbf{R}^{1\times h}, O_t \in \mathbf{R}^{n\times q}, W_{hq} \in \mathbf{R}^{h\times q}$, and $b_q \in \mathbf{R}^{1\times q}$ where q denotes the dimension of the output. X_t represents the input matrix, H_t represents the hidden state, and O_t represents the output matrix where t denotes the current time step. W_{xh} is the input weight matrix, W_{hh} is the previous hidden state weight matrix, and W_{hq} is the output weight matrix. b_h and b_q are the bias terms associated with the hidden layer vector and the output vector, respectively. Figs. 3(b) and (c) show a schematic of the operations conducted on the inputs within a recurrent and output layer, respectively. Within a recurrent layer, the calculation of H_t from one time step to the next is the same as shown in Eq. (13a). Two different weight matrices, W_{xh} and W_{hh}, are generated for the current input and the previous hidden state. An optional bias vector, b_h, can also be included in the H_t calculation. The output unit is analogous to a feed-forward logistic unit in that it contains a single weight matrix with a bias vector as shown in Eq. (13b).

The backpropagation process within RNNs is much more complicated than that within FNNs because of the additional time dimension.

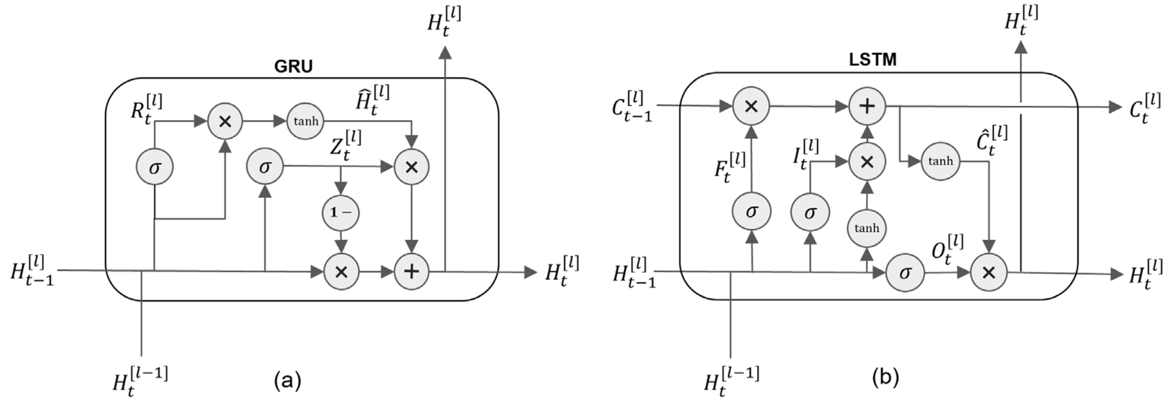


Fig. 4. Schematics of (a) GRU and (b) LSTM.

In order to obtain the exact gradients with respect to all weights and parameters, the computational graph of the RNN needs to be calculated one time step at a time (Werbos, 1990). A problem with this calculation is that, when input sequences are long (>1000), it could result in large matrix products, making the training computationally infeasible in many situations (Zhang et al., 2021a). Another problem associated with long input sequences and gradients is divergent gradient values. As the input sequences become longer and the exponents of the weight matrices increase, the output becomes more likely to be divergent. This phenomenon is often referred to as vanishing or exploding gradients. A solution to this problem is to truncate the gradient at certain time steps to avoid large matrix calculations and divergent eigenvalues. However, truncation may result in loss of relevant information from the early time steps.

4.2.2. Gated recurrent unit (GRU)

Gating in neural networks refers to controlling the expression of key states in neurons. Generally, gating is achieved through the use of the gates which are sigmoid activation functions, $f(x) = \frac{1}{1+e^{-x}}$, that limit the output from 0 to 1. The output is then multiplied by the state variable to control its expression. As mentioned in the previous section, traditional RNN units have difficulties capturing longer-term dependencies due to the phenomenon of exploding or vanishing gradients. GRUs try to alleviate this issue by the addition of reset and update gates (Chung et al., 2014), and are mathematically described by the following equations:

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r) \tag{14a}$$

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z) \tag{14b}$$

$$\hat{H}_t = \tanh(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h) \tag{14c}$$

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \hat{H}_t \tag{14d}$$

where R_t, Z_t, and $\hat{H}_t$ denote the reset gate, update gate, and candidate hidden state, respectively. The subscripts r and z denotes the association of variables with the reset and update gate, respectively, and $R_t \in \mathbf{R}^{n\times h}, Z_t \in \mathbf{R}^{n\times h}, \hat{H}_t \in \mathbf{R}^{n\times h}, W_{xr}, W_{xz} \in \mathbf{R}^{d\times h}, W_{hr}, W_{hz} \in \mathbf{R}^{h\times h}$, and $b_r, b_z \in \mathbf{R}^{1\times h}$. As shown in Eq. (14), the reset and update gates control the formulation of the candidate hidden state, $\hat{H}_t$, and the extent to which to update the hidden state, H_t, with the candidate hidden state. The reset and update gate values are calculated from the current input, X_t, and the previous hidden state, H_{t-1}, with the only difference being different multiplicative weight matrices and bias vectors.

The reset gate aims to capture short-term dependencies by controlling the extent of H_{t-1} that RNN should remember by limiting the expression of $\hat{H}_t$. Therefore, R_t is element-wise multiplied with H_{t-1} to control its expression in $\hat{H}_t$ as shown in Eq. (14c). When $R_t \to 1$, $\hat{H}_t$ will be equal to the hidden state calculation of a traditional RNN unit. When $R_t \to 0$, $\hat{H}_t$ will be equal to the calculations performed within an FNN unit, since there is no information from the previous state.

The update gate aims to capture longer-term dependencies, as it controls the extent to which the new hidden state is a copy of the old hidden state by controlling the ratio between $\hat{H}_t$ and H_{t-1}. The current hidden state, H_t, is simply a weighted average between $\hat{H}_t$ and H_{t-1} controlled by Z_t, as shown in Eq. (14d). When $Z_t \to 1$, H_t ignores $\hat{H}_t$ and subsequently, the current input X_t, resulting in $H_t = H_{t-1}$. In this case, the current hidden state is a direct copy of the previous hidden state. When $Z_t \to 0$, $H_t = \hat{H}_t$, but this does not imply that the previous hidden state is ignored. As seen in Eqs. (14c) and (14d), H_t can still depend on H_{t-1} if $R_t \to 0$. Thus, only when both $Z_t, R_t \to 0$, will the current hidden state solely consider the current input and ignore all previous hidden states.

4.2.3. Long short-term memory (LSTM) unit

GRUs attempt to solve the problem of vanishing and exploding gradients by directly changing the hidden state from one unit to the next. In contrast, the LSTM unit introduces a new state, the memory cell state C_t, to store additional information regarding short-term and long-term dependencies (Hochreiter and Schmidhuber, 1997). As shown by Fig. 4(b), this new memory cell state is passed between LSTM units like a hidden state. In addition, the LSTM unit uses three different gates—the input, forget, and output gates—to dynamically update the values of previous hidden and memory cell states. The motivation behind the memory states and the three gates is similar to that of the GRU, which is to decide how to control the expression of previous states and current inputs in the hidden state.

Similarly to the gates in GRUs, the input, forget, and output gates are all activated by sigmoid functions using the current input, X_t, and the previous hidden state, H_{t-1}. Their mathematical formulations are shown below:

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i) \tag{15a}$$

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f) \tag{15b}$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o) \tag{15c}$$

where I_t, F_t, and O_t denote the input, forget, and output gate, respectively. The subscripts i, f and o denote the association of variables with the input, forget, and output gate, respectively, and $I_t, F_t, O_t \in \mathbf{R}^{n\times h}, W_{xi}, W_{xf}, W_{xo} \in \mathbf{R}^{d\times h}, W_{hi}, W_{hf}, W_{ho} \in \mathbf{R}^{h\times h}$, and $b_i, b_f, b_o \in \mathbf{R}^{1\times h}$. Specifically, the input, forget, and output gates will output values ranging from 0 to 1 to control the expression of key information in the new hidden state.

In an LSTM unit, instead of having a candidate hidden state as in a GRU, a candidate memory cell, $\hat{C}_t$, and memory cell state, C_t, are utilized to capture dependencies in the sequence and are shown by the following equations:

$$\hat{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c) \tag{16a}$$

$$C_t = F_t \odot \hat{C}_{t-1} + I_t \odot \hat{C}_t \tag{16b}$$

where $\hat{C}_t, C_t \in \mathbf{R}^{n\times h}, W_{xc} \in \mathbf{R}^{d\times h}, W_{hc} \in \mathbf{R}^{h\times h}$, and $b_c \in \mathbf{R}^{1\times h}$. The computation of $\hat{C}_t$ is similar to that of the three gates except with a *tanh* activation function that limits the output from −1 to 1 as shown in Eq. (16a). I_t is used to introduce new information to C_t by controlling the expression of $\hat{C}_t$ in C_t. F_t is used to dictate how much old information is to be forgotten from the previous C_t through controlling the expression of C_{t-1} in C_t as shown in Eq. (16b). Finally, as shown in Eq. (17) below, the hidden state, H_t, is the element-wise product between O_t and $\tanh(C_t)$:

$$H_t = O_t \odot \tanh(C_t) \tag{17}$$

where O_t dictates how much of C_t is relevant to the current output and controls the extent of the contribution of C_t to H_t. A difference between LSTM and GRU is that even though the current input information may not be relevant to the current hidden state, this information is still stored in the memory cell state so that it can be used for the computation of future hidden states. In summary, by resolving the issue of vanishing and exploding gradients, the GRU or LSTM units will suppress irrelevant information and better capture longer-term dependencies using a combination of gates. For example, if an earlier input is highly significant for the prediction of future outputs, it is necessary to capture and store this dependency into the hidden state or a separate cell state. In addition, using a vanilla RNN unit without this storage may result in a very large or a very small gradient with respect to the weight matrices associated with the earlier inputs during training, and thus may cause exploding or vanishing gradient when propagated through the layers of the neural network.

4.2.4. Encoder–decoder architecture

One problem with traditional RNN architectures is that they struggle with variable-length input and output sequences (Cho et al., 2014b). In most popular NLP tasks such as language translation, the input and output sequence will likely have different lengths (Sutskever et al., 2014). In the field of time-series forecasting, it can be beneficial to use previous sequences as opposed to a single point to predict a future sequence as certain patterns can be dependent on previous patterns. While the use of GRU and LSTM units alleviate this problem through the usage of memory cells, long-term dependencies are still difficult to capture due to the models' Markov property. Traditional RNN models rely on the previous state to fully capture even earlier states and do not have direct access to those early states. In the encoder–decoder system, the encoder has direct access to all past states within a certain historical window and thus can better capture long-term dependencies. In the context of MPC, the historical window used in calculating the prediction horizon should be tuned to capture different length dependencies within the time-series data. In order to address the aforementioned problems, the encoder–decoder system is designed with two major components: an encoder followed by a decoder. The encoder first takes a variable-length sequence as the input and summarizes it to a context state, which is passed to the decoder. The decoder then maps the context state to a variable-length sequence as the output. The encoder–decoder system can be thought of as a special RNN architecture, since each encoder–decoder unit can be any of the RNN/GRU/LSTM units shown in Fig. 5.

The encoder takes a fixed input sequence with length $M-1$ and summarizes it into a context state that is passed to the decoder. In each encoder unit, each individual input x_i is transformed into the hidden state only and passed to the next encoder unit. As a result of this, the final context state, C, can be thought of as a function of all the previous hidden states within the historical window $C = f(h_{t_{k-M+1}}, \dots, h_{t_k-1})$. The decoder uses the context state and decoder input sequence, $x_{t_k}, \dots, x_{t_{k+N-1}}$, to predict the desired future sequence, $\hat{y}_{t_k}, \dots, \hat{y}_{t_{k+N-1}}$. In the case of time-series forecasting, $\hat{y}_{t_k}$ is generally the predicted states for the next time step at $t = t_{k+1}$.

5. Neural network model construction tutorial

Incorporating machine learning to solve realistic problems is not a straightforward process. It is often a cyclical process that uses model evaluation to iterate between improving the data and improving the model. This cycle is crucial in developing a successful machine learning model, since it evaluates feedback from previous results and implements changes to further improve the earlier steps. A general workflow of the iterative cycle is outlined in Fig. 6. In the following subsections, each step of the proposed workflow is explained in detail, with a focus on developing a neural network-based process model for MPC.

5.1. Problem identification

The first step in developing any ML model is to identify a general problem statement and transform it into a specific ML task. For example, the problem statement might be to optimize the operation of a series of reactors. It is of utmost important to specify if the goal is to optimize a certain chemical species' yield, minimize environmental impact, or maximize overall profits. Even with a well-defined problem statement, there is still no clearly defined ML problem. Following the reactor example, if the goal is to optimize profit, an EMPC can be implemented to control the reactor in real-time to produce at the optimal rate under varying operating costs. For real-time control to be implemented, a dynamic model of the system must be constructed, which is where a neural network model may be considered. In the absence of a traditional first-principles model, a neural network surrogate model is able to provide accurate real-time state information of the reactor comparable to that of the traditional first-principles model (Esche et al., 2022). At this step, the ML task details, such as classification versus regression and supervised versus unsupervised versus semi-supervised, should be formulated based on the task description and data availability. In supervised learning, ML algorithms are trained to learn the target relationship within datasets that contain both the inputs and the labeled outputs. In unsupervised learning, no outputs are given, and ML algorithms attempt to find possible relationships between inputs alone. Finally, in semi-supervised learning, unsupervised learning techniques are incorporated into supervised ML algorithms to avoid the need to label large amounts of data. In the case of most MPC and RTO problems, a regression neural network model trained with supervised learning is sufficient for implementation.

5.2. Data collection

After the ML task is identified, it is important to understand the variety of data sources. This can range from physical devices to network traffic service information. In this work, some of the most common types of data sources will be highlighted to give the reader a general overview, focusing on physical equipment, such as sensors. Sensors are the core data sources for many chemical or manufacturing systems. They aim to measure explicit physical properties, such as temperature, pressure, and flow rate, at different levels of the system (device, subsystems, systems, and environment). Sensors can be classified into two types: direct and indirect. Direct sensors measure the desired quantity directly, for example using a pressure sensor to detect low tire pressure, while indirect sensors measure other quantities and calculate the desired quantity based on the measured quantity, for example, detecting low tire pressure by comparing the relative wheel speed. The

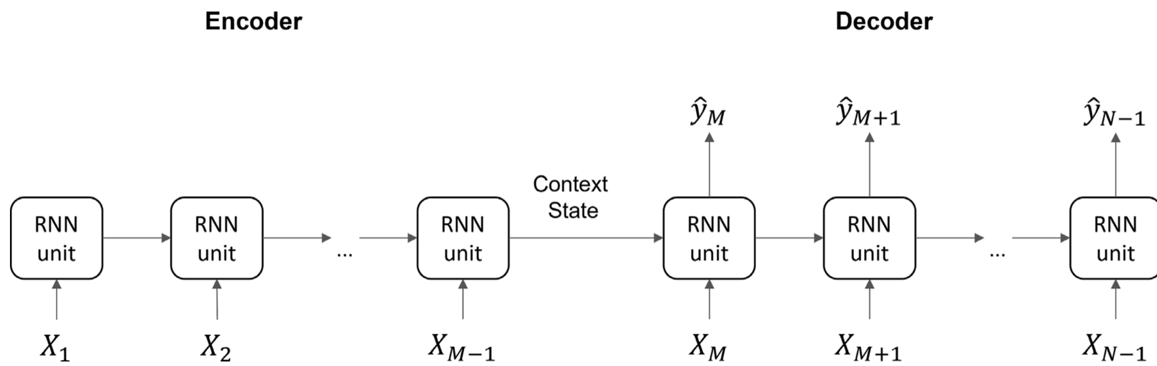


Fig. 5. Structure of an encoder–decoder with RNN units.

Problem Identification
- Physical problem statement
- Formulation of ML task

Data Collection
- Sources of data (e.g. human observation, sensor)
- Data type

Data Processing
- Data cleaning (e.g. structural errors, outlier detection)
- Data transformation (e.g. normalization, scaling)

Model Construction & Training
- Architecture construction (e.g. sparse connection)
- Hyperparameter and loss function
- Training procedures

Model Evaluation
- Testing metrics (e.g. MSE, RMSE)
- MPC integration considerations

Fig. 6. Proposed neural network-based model construction workflow.

major difference is the error and uncertainties produced by the two types of sensors. Indirect sensors are more prone to large uncertainty, as they may involve multiple measurements, and thereby, they may compound the measurement error through the additional calculations. Therefore, depending on the problem statement, levels of uncertainty and error need to be evaluated against the sensors' inherent error to see if the sensor's error range is acceptable. Regardless of how well the ML model is able to reproduce the given dataset, if the data error is too large, applications of the ML model will fail due to the discrepancy between on-line and off-line testing.

When implementing an MPC system, another point to consider is the sampling frequency at which the desired quantity can be measured or the control action can be enacted. In certain situations, the state can only be measured after large time intervals, resulting in a large sampling period with respect to the time constant of the process, which can cause the model to fail to effectively capture the process dynamics

within each sampling period. In addition, due to the sample-and-hold implementation of the MPC, a large sampling period also implies that the MPC may be activated less frequently than the dynamics of the process, leading to performance deterioration. Therefore, the sampling frequency of states and the process behavior must be considered in the design of an MPC. An example of the above concern is the use of gas chromatography to measure the composition within a chemical reactor because the measurement and subsequent analysis may take more than ten minutes to report a concentration, which does not provide sufficient real-time concentrations for certain processes with fast dynamics.

Data collected from the system may come in a variety of different forms. Even when measuring the same physical property, the data can come in both structured and unstructured forms. For example, a sensor can produce time-series temperature data in the form of a numerical list or table, which is considered structured data. Alternatively, infrared cameras may generate a heat map that may be more relevant than time-series data but in an unstructured form. In addition, the data preprocessing step may vary as a result of the state of the process—static versus dynamic. In the chemical engineering context, static data can be thought of as steady-state, whereas dynamic data refers to transient behavior. For dynamic data, the order may be important to the process, whereas static data can be shuffled. Whether it is structured/unstructured or static/dynamic data, identifying the data type and quality is of utmost importance to ensure the success of the ML model.

5.3. Data preprocessing

Data preprocessing is an essential step to convert the raw data collected from sensors and experiments to a clean and usable dataset (e.g., Modified National Institute of Standards and Technology (MNIST), UCI official dataset). The amount of work necessary to transform raw data to a usable dataset is often overlooked. A clear overview of data preprocessing steps for machine learning is demonstrated in Kotsiantis et al. (2006). Specifically, the first step is to remove duplicates and irrelevant observations from the raw dataset, which is almost inevitable in large datasets. In a manufacturing setting, multiple sensors are implemented but they may not all be relevant to the ML task. The main goal of data preprocessing is to reduce the size of the dataset while maintaining all the relevant information. The next step is to look for structural errors such as corrupted data values and missing or mislabeled features. The process of filling in missing data values is called data imputation (Zhang et al., 2006), which can range from simply using the mean, median, or mode of the column to implementing simple machine learning techniques such as k-nearest neighbors. Interpolation methods can also be used depending on the nature of the dataset. Finally, after the dataset is treated for missing and irrelevant information, outlier detection methods can be implemented to increase the performance of the ML models. Common outlier detection methods include Z-score (Habib et al., 2015), probabilistic models, and clustering methods. For parametric datasets, which are datasets with a known distribution, the Z-score method proves to be an efficient way to eliminate outliers and is shown below:

$$z = \frac{x - \mu}{\sigma} \tag{18}$$

where x is the current data point, μ and σ are the mean and standard deviation within the dataset, and z is the Z-score for the data point. As shown in Eq. (18), Z-score uses the mean, μ, and the standard deviation, σ, to assess whether a data point is an outlier or not. The common Z-score threshold is around $\pm 2.5 \sim 3.5$. Clustering methods can be used to detect outliers for non-parametric datasets, and specifically, the density-based spatial clustering of applications with noise (DBSCAN) method is particularly effective. DBSCAN focuses on finding neighbors by density on an "n-dimensional sphere" with radius, ϵ (Ester et al., 1996). DBSCAN aims to cluster all the outliers to an out-of-bound cluster where they can be further processed or removed.

In addition to off-line outlier detection methods, on-line outlier detection algorithms have been proposed to deal with dynamic time-series data. Liu et al. (2004) have proposed an algorithm that combines an autoregressive moving average process model with a modified Kalman filter that uses past and current data to estimate the current data point and its variation. Kieu et al. (2018) have proposed the use of deep neural networks in detecting outliers within time-series data. Specifically, the raw time-series data is enriched with more statistical features and an autoencoder is used to select the most representative statistical features. Outliers often have non-representative features and thus deviations from the enriched time-series data is taken as outliers. Finally, Li et al. (2019) have used generative adversarial networks (GAN) for anomaly detection in time-series data. The proposed GAN framework uses LSTM units as the basis for its discriminator and generator to capture the temporal correlation in the time-series dataset. The discriminator itself is a direct tool for anomaly detection. The generator is exploited to capture the mapping from the latent space to the real data distribution, and this distribution can be used to detect anomalies in the test dataset. A combination of the discriminator and generator aspect of the proposed GAN is used as a metric to classify outliers within time-series data.

For any ML project, it is important to have at least two sets of data for training and testing purposes. In the case of neural network models, a third validation split is also recommended for model tuning. A typical split ratio is 80/20 or 70/15/15 with validation, although these ratios can and should be tuned based on the problem statement and availability of data. The training dataset is used to adjust the weights and biases of the neural network, the validation dataset is used to adjust the hyperparameters (number of neurons, number of layers, etc.) of the neural network, and the testing dataset is used to evaluate the performance of the neural network to an unseen dataset. Data splitting is conducted before any further data processing to prevent data leakage. Data leakage refers to the leakage of information, such as the mean or standard deviation of the testing to the training dataset, that can affect the testing accuracy. The next step in data preparation is data processing, which refers to the application of different transformations to the dataset to improve training performance. Data scaling, which applies some type of scaler to normalize the dataset within a certain range, can be applied to both structured and unstructured data, making it commonly the first transformation applied to a dataset. This prevents large discrepancies in the gradient between different input features during model training, which can cause weight values to change dramatically, making the training process unstable. The two most commonly used scalers are the Min–Max scaler and the Z-score scaler. A min–max scaler, shown in Eq. (19) below, scales the entire dataset's values between the user-defined feature range and is calculated using the following equation:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}(f_{max} - f_{min}) + f_{min} \tag{19}$$

where x is the current data point, x_{max} and x_{min} are the maximum and minimum values within the dataset, f_{max} and f_{min} are the user-defined maximum and minimum feature values, and x_{scaled} is the min–max scaled data point. Generally, f_{max} and f_{min} are 1 and 0 by default, respectively, which results in x_{scaled} to be within the range of 0 to 1. On the other hand, the Z-score scaler, shown in Eq. (18), scales the dataset to a zero-mean distribution with unit standard deviation. After data scaling, more task-specific transformations can be implemented to generate the input tensor. Based on the type of ML library (Tensorflow, Keras, Pytorch) and the type of neural network chosen (FNN, RNN), the input shape of the training data is processed differently. Custom helper functions can be implemented to transform the original dataset into the desired training shape.

Remark 3. For time-series forecasting tasks, data cannot be entered into the model for training as a single sequence. For the training of

neural networks, the input sequence needs to be partitioned into fixed length intervals of historical and prediction sequences. In other words, the neural network training sequence would have a total length of $M + N$ with the input being the historical window plus the instantaneous measurement and the output being the prediction sequence of length N. A common algorithm to achieve this sequence format is the sliding window algorithm (Dietterich, 2002). In the sliding window algorithm, criteria for the desired window are defined, and the window is slid across the entire sequence with a fixed step size. In this scenario, the criteria are that the total window length is equal to $M + N$ and less than the total sequence length.

5.4. Model construction

Depending on the ML task, there are various models that can be applied. As mentioned previously, the focus of this article will be on supervised regression models for dynamic data with consideration for control purposes. A variety of different models are compared with clearly defined trade-offs in performance, interoperability, and computational cost.

5.4.1. Model architecture

Feed-forward Neural Networks (FNN)

Several different neural network structures were introduced in Section 4, and all of them can be applied to modeling nonlinear dynamic systems depending on the specific process and computational resources available. When constructing the process model for MPC, it should be noted that the MPC does not require the entire prediction sequence within one sampling period from the process model; as a minimum, only the last time step output, $t = t_{k+N}$, is required, where $t_{k+N} - t_k$ corresponds to one sampling period Δ in the MPC formulation. Therefore, the simplest case of a process model is a MIMO FNN model with the input being the process states and manipulated input at the current time step, $t = t_k$, and the output being the process states at the end of the prediction sequence, $t = t_{k+N}$. Note that this time step is analogous to the integration time step of numerical integration methods when simulating a system of ODEs. A variation of this formulation is the addition of the previous time steps as input to the FNN as shown in Mohanty (2009). With this formulation, the FNN will take in M different time steps of process states and manipulated inputs to predict the process states at $t = t_{k+N}$. As proposed in Kittisupakorn et al. (2009), another method is to train an FNN that only predicts a single time step ahead; this FNN can be called repeatedly for N times, or N such FNNs can be trained and stacked together, to obtain the process states at time step $t = t_{k+N}$. This iterative method is similar to the calculations done in RNNs but without the feedback/recurrent connections within individual logistic units. It is important to note that it is beneficial to use a framework that is able to predict intermediate states rather than only the last states within each sampling period. This is because intermediate states provide more information on the predicted state trajectory and provides a better representation of the cost function of the MPC optimization problem, since the cost function is typically in the form of an integral over the entire prediction horizon. Having access to intermediate states with a shorter time interval in between also allows for better numerical approximations that are necessary within the MPC optimization.

Recurrent Neural Networks (RNN)

While directly implementing a MIMO FNN can satisfy the information needed by the MPC to calculate the optimal control trajectory, this approach may not result in the best performance due to the loss of intermediate information. Therefore, RNNs can be used in place of FNNs as capturing the ordinal aspect of dynamic dataset improves the performance since information between the first and last prediction time steps may have a significant impact on each other (Mohajerin and Waslander, 2019). Based on the iterative FNN model, an improvement is to replace the feed-forward logistic units with recurrent units. Specifically, recurrent units will be used to account for the N future time steps at which the process states will be predicted i.e., every intermediate time step between $t = t_k$ and $t = t_{k+N}$. In addition, different types of recurrent units, such as LSTM or GRU, can be used in place of normal recurrent units to improve performance for certain processes. Zarzycki and Ławryńczuk (2021) have compared the performance between GRU and LSTM as process models in MPC for a chemical reactor system. Both performed better than regular recurrent units, and GRU was recommended due to its lower computational cost. It is important to note that the choice between recurrent units is highly process and data dependent.

Other than using different types of neural network, the specific connections within a neural network can also be customized to improve performance, interoperability (Bonassi et al., 2022), and computational efficiency (Xu et al., 2021). Wu et al. (2020) have developed a physics-based RNN model with knowledge of the structure of the process. Specifically, for two CSTRs in series, two partially-connected LSTM layers were used to model the input and output connections of each CSTR. Compared to the fully-connected LSTM model, the partially-connected model allowed for the decoupling between the second CSTR's input from the first CSTR's process states, which was also reflected in the real system. From a control perspective, the decoupling effect simplifies the system by reducing the interactions between the control actions resulting in faster convergence and better stability. Alhajeri et al. (2022) have performed a comparative study on RNNs that incorporate physical knowledge and traditional fully-connected RNNs in the context of a large-scale complex chemical process simulated with Aspen Plus Dynamics. The physics-based RNNs displayed superior accuracy and computational efficiency compared to the fully connected RNNs, which resulted in better convergence speed and stability in MPC integration.

Encoder–decoder Neural Networks

For certain dynamic processes, such as HVAC systems, long-term dependencies and past sequential trends are particularly important to predicting future process states. Therefore, a special architecture of RNNs, encoder–decoder RNNs, can be used to address the above problems. Compared to the input of traditional RNN structures, the input of the encoder–decoder system will be the sequence from $t = t_{k-M+1}$ to $t = t_k$ rather than the current time step $t = t_k$. The encoder then summarizes all the historical information within time period, $t = t_{k-M+1}$ to $t = t_{k-1}$, into a single context state, which is used in the decoder as the initial hidden state. Additionally, the decoder takes input at the current time step, $t = t_k$, and additional inputs at future time steps, $t = t_{k+1}$ to $t = t_{k+N-1}$, if they are available and relevant to the prediction process. The implementation of the encoder–decoder system may vary depending on the ML library used, but a general pseudocode structure is shown in Pseudocode 1. In Pseudocode 1, by defining $return_sequence = True$, the RNN unit will return an output at every time step rather than only once at the end of the sequence. Similarly, by defining $return_state = True$, the RNN unit will return all states in addition to the final hidden state. In the case of a LSTM unit, the cell state can also be accessed through the $return_state$ argument. Zhang et al. (2021b) have demonstrated that for air pollution data that contain many long-term dependencies, encoder–decoder-based RNNs perform better than LSTM/GRU-based RNNs. One thing to note is that encoder–decoder RNNs typically use more computational resources during both training and inference, and for most dynamic processes, an LSTM or GRU is sufficient to capture the system behavior. Therefore, it is important to start with a simple baseline model and build up the complexity of the neural network models. Finally, as there always exists a trade-off between model performance and computational efficiency, it is recommended to clearly track each model's accuracy and complexity after training.

Pseudocode 1: Encoder–decoder Architecture

Encoder
{
  Input
  {
    layer class: input layer
    shape: (Batch size) × (Encoder Length) × (Number of States and Inputs)
  }
  (*input: encoder_input_data*)
  output: encoder_input
  Recurrent Layers (can be repeated)
  {
    layer class: LSTM or GRU
    hidden units: n_encoder
    return sequence: False
    return state: True
  }
  (*input: encoder_input*)
  output: encoder_state_h, encoder_state_c
}

Decoder
{
  Input
  {
    layer class: input layer
    shape: (Batch size) × (Decoder Length) × (Number of States and Inputs)
  }
  (*input: decoder_input_data*)
  output: decoder_input
  Recurrent Layers (can be repeated)
  {
    layer class: LSTM or GRU
    hidden units: n_decoder
    return sequence: True
    return state: False
  }
  (
    input: decoder_input
    initial state: encoder_state_h, encoder_state_c
)
  output: decoder_sequence
  Dense Layers (can be repeated)
  {
    layer class: Dense
    hidden units: n_decoder
  }
  (*input: decoder_sequence*)
  output: prediction_sequence
}

Remark 4. Instead of constructing a single neural network model, multiple models can be created from the same dataset and used together to predict the future process evolution. This is also known as ensemble learning and is known to have several advantages over using a single neural network model. First, ensemble learning allows for more generalization and prevents overfitting as the algorithm can be exposed to different subsets of the dataset through n-fold cross-validation methods. In other words, ensemble learning can reduce the variance of the algorithm while maintaining a low bias for individual models (Polikar, 2012). Second, due to the nonlinearity of the neural network models, the optimization associated with the process model is non-convex and is an NP-hard problem. Therefore, through the use of different weight initialization methods, the neural network model can potentially avoid getting trapped at local minima and arrive at more optimal sets of weights that allows the neural network to accurately approximate the latent function that transforms input sequences to corresponding output sequences (Wu et al., 2019b). Third, uncertainty during model selection can be better accounted for using ensemble learning algorithms than individual learning algorithm (Mendes-Moreira et al., 2012).

There are multiple ways to construct an ensemble learning algorithm. In particular, two general categories, homogeneous and heterogeneous, will be discussed in this paper. Homogeneous ensemble learning algorithm consists of models with a single base learning algorithm in which individual models are trained from different subsets of the given dataset through methods such as n-fold cross validation and bootstrap sampling. In n-fold cross validation, n different training and testing dataset splits are conducted and each dataset split is used to train a different model (Donate et al., 2013). In bootstrap sampling, also commonly known as bagging, several bootstrap datasets take samples from the initial training set without replacement. As a result, multiple bootstrap training sets are constructed in which the initial training set may appear once, more than once, or may not appear at all (Domingos, 1997). Individual learning models are then trained using the different datasets and a weighted average is used to integrate the individual models together. Conversely, heterogeneous ensemble learning refers to the compilation of different learning algorithms from the same dataset. Specifically, multiple machine learning methods, such as FNN, RNN, support vector regressors (SVR), and gradient boosting machines (GBM) can promote better diversity within the ensemble learning algorithm and thus improve performance (Zefrehi and Altınçay, 2020; Ribeiro et al., 2020).

5.4.2. Hyperparameter tuning

Another important aspect of model development is the tuning of model hyperparameters. In machine learning, hyperparameters refer to parameters that are defined by the user rather than those optimized during training. These can include the number of neurons and layers, the optimizer, the mini-batch size, and others.

The number of layers and neurons is often very important to the model performance as it defines the size and complexity of the model. Depending on the complexity of the input–output relationship, the size of the neural network should be constructed appropriately. While increasing the number of layers and neurons will increase the training performance, it will also make the model less generalizable, which is also known as overfitting. On the contrary, using an overly simplified model to model a complex process leads to not capturing input–output relationships, which is a form of underfitting. Methods to recognize overfitting and underfitting will be explained more in detail in the model training section.

In addition to changing the size and complexity of the neural network, the choice of optimizer is also very important for the final performance of the model. Three popular optimizers, stochastic gradient descent (SGD) with momentum, RMSprop, and *Adam*, are shown in Eqs. (20a), (20c), and (20e), respectively, below:

$$w := w - \alpha m \tag{20a}$$

$$m = \beta m + (1-\beta)\nabla E \tag{20b}$$

$$w := w - \frac{\alpha}{\sqrt{v+\epsilon}}\nabla E \tag{20c}$$

$$v = \gamma v + (1-\gamma)\nabla^2 E \tag{20d}$$

$$w := w - \frac{\alpha}{\sqrt{v+\epsilon}}m \tag{20e}$$

where w, m, and v are the weight, the momentum of the gradient, and the moving average of squared gradients, respectively, and are updated

at every epoch. β and γ are hyperparameters for momentum and adaptive learning rate, respectively, ϵ is a sufficiently small constant (10^{-8}), and E is the cost function. In Section 4.1, the most fundamental and core ML optimization strategy, gradient descent, is shown in Eq. (8). GD updates the weights and biases after every epoch which means the full gradient is calculated for all observations at each epoch. Although accurate, calculation of the full gradient for every epoch is slow and can cause long computation times for large datasets. SGD offers a solution for this problem by approximating the full gradient through calculating the gradient in mini-batches. Subsequently, a momentum term m, as shown by Eq. (20b), can be added to the SGD algorithm to accelerate the rate in which gradients move, leading to even faster convergence as shown in Eq. (20a). Up to this point, when the weights are updated, the learning rate, α, is constant for all weights. However, the magnitude of the gradient can be different for different weights, which leads to the creation of an adaptive learning rate. Thus, RMSProp, shown in Eq. (20c), adds a square root term of the square of the gradient v, where v is calculated in Eq. (20d). Finally, the aforementioned two methods, adaptive learning rate and momentum, can be combined into one which is the *Adam* optimizer, as shown in Eq. (20e). Kingma and Ba (2014) has demonstrated that the *Adam* outperforms other stochastic optimization methods and works well on practical datasets in most situations.

Remark 5. The mini-batch size refers to how many observations are used when calculating the gradient and updating the weights. Learning rate is another important hyperparameter. Learning rates that are too small lead to slow convergence of the model, but larger learning rates may lead to missing the optimal solution. Therefore, adaptive learning rate algorithms, such as *Adam*, are highly recommended.

Several searching algorithms can be applied to conduct hyperparameter tuning. The most intuitive method is grid search in which only one hyperparameter is varied at a time to evaluate the change in model performance with respect to the varied hyperparameter. While grid search is comprehensive, the computational cost of grid search is immense when associated with a high dimensional hyperparameter space. Therefore, random search was developed to reduce the large computational cost of grid search through randomly selecting combinations of hyperparameters instead of enumerating all possible combinations (Bergstra and Bengio, 2012). Random search greatly outperforms grid search when only a small amount of hyperparameters affect the model performance. In addition to random search, Bayesian-based and gradient-based hyperparameter optimization methods have also been introduced (Snoek et al., 2012; Maclaurin et al., 2015). Manually implementing a hyperparameter search algorithm is labor and skill intensive. Therefore, commercial services, such as Google's HyperTune, are candidate options for conducting hyperparameter tuning.

5.4.3. Cost functions and regularization

In Section 4, a generic cost function is shown in Eq. (7). The cost function is the mean of all errors within a dataset while the loss function refers to the error of a single datapoint. For regression tasks, the absolute error and the square error are good choices for the loss function within the cost function. For certain logarithmic datasets, the square logarithmic error can also be used if the data points cannot be scaled during the data preparation step. Since the cost function considers the entire dataset, the mean of the loss of all the individual data points is taken as the cost function, leading to the terminology of mean absolute error (MAE), mean square error (MSE), and mean square logarithmic error (MSLE). Each of the aforementioned loss functions can be transformed into their respective cost function using the following equations:

$$E_{MAE} = \frac{1}{R}|Y - \hat{Y}|_1 \tag{21a}$$

$$E_{MSE} = \frac{1}{R}|Y - \hat{Y}|^2 \tag{21b}$$

$$E_{MSLE} = \frac{1}{R}|\log(Y+1) - \log(\hat{Y}+1)|^2 \tag{21c}$$

In addition to selecting a loss function, a regularization term can be added to the cost function for smoothing purposes and to prevent overfitting. Three of the most common regularization methods, L_1, L_2, and Elastic net, are shown below:

$$E_{L_1} = \frac{1}{R}\sum_{i=1}^{R} L(y,\hat{y}) + \lambda|W|_1 \tag{22a}$$

$$E_{L_2} = \frac{1}{R}\sum_{i=1}^{R} L(y,\hat{y}) + \lambda|W|^2 \tag{22b}$$

$$E_{L_{elastic}} = \frac{1}{R}\sum_{i=1}^{R} L(y,\hat{y}) + \lambda\left(\theta|W|_1 + \frac{1-\theta}{2}|W|^2\right) \tag{22c}$$

where λ is the regularization hyperparameter and θ is the Elastic net hyperparameter. Both L_1 and L_2 regularization methods are based on the concept of using different L_p norms to penalize high regression coefficients for complex models to avoid overfitting. L_1 regularization, or Lasso regression, penalizes the absolute value, L_1 norm, of the weights as shown in Eq. (22a). L_1 regularization is generally used for sparse feature sets, since it can perform feature selection by zeroing out irrelevant features' weights. L_2 regularization, or Ridge regression, penalizes the square, L_2 norm, of the weights as shown in Eq. (22b). L_2 regularization cannot eliminate features due to the nature of the L_2 norm, but performs better than L_1 regularization in most cases. Elastic net, shown in Eq. (22c), linearly combines L_1 and L_2 regularization through a weighted sum and adds an additional hyperparameter, θ, to adjust the ratio between the two methods. In the aforementioned regularization methods, a regularization parameter λ is included to control the extent of regularization within models. Elastic net outperforms both regularization methods in most situations, especially when the number of features is much larger than the number of observations (Zou and Hastie, 2005).

5.5. Model training

During model training, it is important to save the training history of the model. The training history can indicate whether the model has experienced trends of over- or under-fitting. The most common plot shows the evolution of the loss function with respect to epochs during training and validation. In most cases, longer training will result in model overfitting to the training dataset and will not generalize well to the validation and testing datasets. In the case of overfitting, the training loss keeps on decreasing while the validation loss increases rather than decreases as more epochs pass. An early stopping function can be implemented to prevent excessive training through the monitoring of validation loss. For example, an early stopping criteria of a certain number of consecutive increasing validation loss can be defined and the training process will stop if the criteria is reached. On the contrary, in underfitting, both the training and validation loss is very high and is still decreasing at an exponential rate. Ideally, a good model should have similar training and validation loss at the end of training. A good application to keep track of all training data is Tensorboard which automatically plots all training and validation curves as well as model structure graphs.

Remark 6. In realistic processes, noise is inevitably present due to a combination of process disturbances and sensor errors. In the earlier data preparation step, outlier detection techniques, such as DBSCAN, are introduced to eliminate anomalies. However, a majority of the noise is within operating range and inherent to the data points. Many different training routines are developed to combat noisy datasets. The dropout method is developed to prevent overfitting in large neural networks through randomly dropping connections to neurons during training (Srivastava et al., 2014). Dropout has then been extended to

RNN models and adapted with Monte Carlo method to handle noisy data during training (Gal and Ghahramani, 2016; Wu et al., 2021a). In addition to the dropout method, Han et al. (2018) has developed a new training method called "Co-teaching" to effectively train robust deep neural networks. The "Co-teaching" method trains two neural networks in parallel and allows the exchange of information during each mini-batch. During forward propagation, each neural network selects a subset of assumed "clean" data and sends them to the other network. During backpropagation, each neural network updates its weights using only the "clean" data sent by the other network. Wu et al. (2021a) has applied both of these techniques to an example of a chemical process with the integration of MPC. It is demonstrated that both methods display superior performance compared to the baseline model in fitting and MPC set-point convergence.

Remark 7. Up to this point, only neural networks trained from a fixed dataset have been discussed. In reality, a process may change over time due to a mixture of external (e.g., equipment degradation and disturbances) and internal (e.g., fouling within equipment) factors. Therefore, in the presence of model uncertainty and parameter variations, on-line adaptive learning is essential to maintaining an accurate and up-to-date process model. At the same time, the frequency with which process models are retrained needs to be limited by event-triggered schemes in order to improve applicability and efficiency of adaptive control systems (Tabuada, 2007; Wang and Lemmon, 2008). In Hedjar (2013), a neural network identifier has been trained to detect process variations and update the parameters of the process model. Alanqar et al. (2017) has proposed an adaptive EMPC system through the use of an error-triggered model re-identification scheme in which a threshold is set between the predicted and measured states as a trigger for model update. Building on top of the previous works, Wu et al. (2019a) has developed a dual event and error-triggered on-line update scheme for RNN models in a Lyapunov-based MPC. Specifically, the event-triggered model re-identification occurs when a triggering condition based on state measurements is violated, while the error-triggered update scheme is activated when the accumulated RNN modeling error exceeds some error threshold. With the proposed framework, the adaptive RNN-based LMPC performs better than a standard RNN-based LMPC in terms of guaranteed stability and control action smoothness.

5.6. Model evaluation

After the machine learning model has been constructed and trained, it is necessary to evaluate the trained model with meaningful metrics. In this section, general machine learning metrics are discussed as well as specific errors for MPC. Previously, the training and validation errors were used to check the training process and tune the model parameters. In model evaluation, the testing dataset is used to evaluate the finalized model's performance in terms of both accuracy and generalization on a set of unseen data. In the context of process modeling, the testing dataset may contain operating conditions unseen in the training operating conditions. For time-series forecasting tasks, general regression testing metrics that can be used include MAE, MSE, mean average percentage error (MAPE), and root mean square error (RMSE). These error metrics describe how close the model's predicted states are to the true process' states. Hence, errors are used to check the training process and tune the model parameters. Specifically, it is necessary to ensure that the training error is below a certain bounded modeling error threshold in order to guarantee exponential stability for the nominal system of Eq. (1) under a Lyapunov-based controller built using an RNN model. Subsequently, the generalization of the model needs to be tested using a set of unseen data. In the context of a specific process, the testing dataset may contain some unseen operating conditions. For time-series forecasting tasks, general regression testing metrics can be used that include MAE, MSE, mean average percentage error (MAPE), and root mean square error (RMSE). These error metrics describe how well the model predicts the future process states. It has been demonstrated through simulations that lower testing metrics lead to improved stability in the control system. However, there are also other aspects that should be considered in the context of integrating control. For example, the smoothness of the prediction trajectory is very important for the stability the control action implemented.

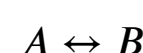
Remark 8. A more generalized error bound for RNN can be calculated using statistical machine learning theory. It is a measure of how well a neural network hypothesis learned from training data generalizes to unseen data so that it is more comprehensive for a wide range of operating conditions rather than the given training and testing conditions (Wu et al., 2021b, 2022). The generalization error bound is found to be such that it does not have any dependency on the states of the neural network, as it is dependent on the weights, sample size, length of the input sequence, and width and depth of the neural network.

6. Neural network-based MPC implementation: Chemical process example

In this section, a nonlinear chemical process is used to demonstrate the performance of various LMPCs. In particular, three different neural network models (FNN, RNN, and encoder–decoder) are considered as the process models for LMPCs. The specific chemical reactor example has been chosen to be tractable in terms of the understanding of its dynamic behavior and evolution by chemical engineers and researchers familiar with chemical processes.

6.1. CSTR process description

Consider a non-isothermal, well-mixed CSTR, where the following reversible first-order exothermic reaction is taking place:



The following mass and energy balance equations represent the first-principles model that describes the process dynamics:

$$\frac{dC_A}{dt} = \frac{1}{\tau}(C_{A0} - C_A) - r_A + r_B \tag{23a}$$

$$\frac{dC_B}{dt} = \frac{-1}{\tau} C_B + r_A - r_B \tag{23b}$$

$$\frac{dT}{dt} = \frac{1}{\tau}(T_0 - T) + \frac{-\Delta H}{\rho C_p}(r_A - r_B) + \frac{Q}{\rho C_p \upsilon} \tag{23c}$$

$$r_A = k_A e^{\frac{-E_A}{RT}} C_A \tag{23d}$$

$$r_B = k_B e^{\frac{-E_B}{RT}} C_B \tag{23e}$$

where the notations C_A and C_B represent the concentration of chemical A and B, respectively. The reactor temperature is denoted as T, the inlet temperature is denoted by T_0, and C_{A0} is the inlet concentration. Also, for the reaction kinetics, the reaction rate constant and the activation energy for the forward reaction are denoted as k_A and E_A, respectively, and k_B and E_B for the backward reaction. The residence time of the reactor is τ, the heat capacity of the liquid mixture is denoted by C_p, the volume of the reactor is denoted by υ, and the reaction enthalpy is ΔH. The CSTR is surrounded by a heating/cooling jacket that provides/removes heat at a rate Q to/from the reactor. The optimal steady state point for the process described in Eqs. (23). Table 1 lists the values of the process parameters along with the steady-state values at the optimal operating point.

The control objective in this example is to drive the process states, C_A, C_B, and T, to the optimal steady-state point by manipulating the heating rate Q and the inlet concentration C_{A0}. The process variables are all considered in deviation form from their steady-state values, which gives $x^T = [x_1, \ x_2 \ , x_3] = [C_A - C_{A_{ss}} \ , C_B - C_{B_{ss}} \ , T - T_{ss}]$ such that the origin is the equilibrium point of this system. Furthermore, this extends to the manipulated inputs $u = [u_1, u_2] = [Q -$

Table 1
Parameter and steady-state values for the CSTR.

$C_{A_{ss}} = 0.4977$ mol/L	$\tau = 60$ s
$C_{B_{ss}} = 0.5023$ mol/L	$Q_{ss} = 40386$ cal/s
$C_{A0_{ss}} = 1$ mol/L	$V = 100$ L
$T_o = 400$ K	$T_s = 426.743$ K
$k_A = 5000$ /s	$E_A = 1 \times 10^4$ cal/mol
$k_B = 10^6$ /s	$E_B = 1.5 \times 10^4$ cal/mol
$R = 1.987$ cal/(mol K)	$\Delta H = -5000$ cal/mol
$\rho = 1$ kg/L	$C_p = 1000$ cal/(kg K)

Q_{ss} $, C_{A0} - C_{A0_{ss}}]$, where the control action u is bounded by a lower bound $u^{LB} = [-40,000\ cal/s,\ -1\ mol/L]$ and an upper bound $u^{UB} = [40,000\ cal/s,\ 2\ mol/L]$.

The nonlinear minimization problem of LMPC is resolved using the interior point optimizer (IPOPT) package for each sampling time, which is $\Delta = 10\ sec$ with $h_c = 0.5\ sec$. IPOPT is an open source optimization package that can be employed to solve nonlinear optimization problems (Biegler, 2010). It employs an interior point-line search filter method, which tries to find a local optimum. Furthermore, the objective function of the LMPC can be formed as $L(x,u) = x^T Ax + u^T Bu$, where A and B are diagonal penalty matrices. Matrices A and B are critical for the MPC performance and are tuned according to the guidelines discussed in Alhajeri and Soroush (2020). The chosen Lyapunov functions are $V(x) = x^T Px$, where $P = diag\{10^5, 10^5, 1\}$ is a positive definite matrix.

6.2. Data generation and processing

In this work, extensive open-loop simulations is conducted using the first-principles model to build the training and testing dataset. Specifically, starting from various initial conditions, the process model is integrated using the explicit Euler method under time-varying manipulated inputs while recording the states' evolution at each time step (i.e., $x_{t_{k+1}}, \dots, x_{t_{k+N}}$, where $N = \Delta/h_c$). Subsequently, the dynamic time-series data are transformed into the format required for training and testing through the use of the sliding window algorithm. In our case, the criteria for the sliding window is that the total window length is equal to the sum of the input window length, M, and prediction horizon length, N. The open-loop simulation runs for 50 time steps for 432 different initial conditions, and the first 30 time steps are used as the training dataset, while the remaining 20 time steps are used as the testing dataset. In total, 43200 data samples were available for training and testing. The goal is to identify whether the model can learn the behavior of the process within the first 30 time steps to predict the next 20 time steps starting from different initial conditions. Both the training and testing datasets are scaled with respect to themselves only to avoid information leakage. The training and testing datasets are scaled from 0 to 1 using Eq. (19) to avoid large discrepancies between gradients during training. Finally, the training and testing datasets are reshaped into 3-D tensors with input and output dimensions of $(R_{train}, M + N, n_{states} + n_{inputs})$ and $(R_{test}, M + N, n_{states})$, respectively, where R refers to the total number of training/testing observations and n_{states} and n_{inputs} refers to the number of states and inputs, respectively.

6.3. Neural network models construction

In this work, the performance of using three different types of neural network models: FNN, RNN with LSTM units, and encoder–decoder with LSTM units is investigated. A general hyperparameter search is conducted on the hidden layers, hidden units, and activation functions for each of the neural network models. The final FNN is constructed with a single fully-connected layer with 10 hidden units and a linear activation function. The RNN consists of a single LSTM layer with eight hidden units and one fully-connected output layer on top with N hidden units. The LSTM layer and fully-connected layer have activation functions of *tanh* and linear, respectively. The encoder–decoder model consists of two LSTM layers (one encoder and one decoder layer) connected at the ends with 8 hidden units each and a single fully-connected output layer on top of the decoder LSTM layer with N hidden units. The LSTM layers and fully-connected layer have activation functions of exponential linear unit (ELU) and linear, respectively. Due to the different model architectures, the inputs to the models are slightly different. In FNN, the time step t_k will be used to predict the time step t_{k+N}. In RNN, the time step t_k will be used to predict the future sequence $t_{k+1}, \dots, t_{k+N}$. In the encoder–decoder system, the historical and present information $t_{k-M+1}, \dots, t_k$ will be used to predict future time steps $t_{k+1}, \dots, t_{k+N}$. All these models are comparable since, in our MPC implementation, only the last time step, t_{k+N}, is used in the objective function to determine the optimal control trajectory for one Δ. The neural network model will be called K times to determine the control trajectory for the full MPC prediction horizon.

For training, an MSE cost function is used, and each model is trained for 100 epochs with a callback function that would follow an early stopping criterion. Each model's training and validation errors are logged to ensure no overfitting or underfitting occurs. From the training and validation errors, the FNN model is rejected because its MSE is two orders of magnitude larger than the other two models. Therefore, the FNN is not tested in both open and closed-loop simulations to save computational resources due to its poor training performance.

Remark 9. The architecture and training procedures of the three models are kept simple because of the good quality of the simulation dataset and the large number of available operating conditions. In real scenarios, data may be contaminated with noise or be insufficient in quantity, and thus different architectures or processing steps need to be explored to combat these problems. Some methods for combating noisy data are included in Section 5.4.

Table 2
MSE comparison of the open-loop prediction results between the neural network models and the first-principles model.

	$C_A - C_{A_{ss}}$ [mol/L]	$C_B - C_{B_{ss}}$ [mol/L]	$T - T_{ss}$ [K]
LSTM	3.24×10^{-6}	1.00×10^{-6}	3.61×10^{-2}
Encoder–decoder	5.29×10^{-6}	7.84×10^{-8}	1.69×10^{-2}

6.4. Open-loop performance

Subsequently, after the development of different NN models, an open-loop simulation was performed to evaluate the generalization of the models and their ability to capture the dynamics of the given CSTR process. During open-loop simulation, the two manipulated inputs, the heating rate Q and the initial concentration C_{A0}, were varied to compare the states predicted by the models versus the ground truth given by the first-principles model. Fig. 7 illustrates the open-loop prediction using the LSTM model and the encoder–decoder model in response to time-varying inputs. Three different combinations of $[Q - Q_{ss}, C_{A0} - C_{A0_{ss}}]$ with values of $[1.5\times10^4, 0]$, $[3\times10^4, 1.5]$, and $[2\times10^4, 1]$ were introduced in the forms of two step changes at $t = 100$ sec and $t = 200$ sec. The two models' trajectories from the plot are in good agreement with the first-principles model. Specifically, the MSE for each state given by the LSTM and encoder–decoder models in the open-loop simulation is computed and shown in Table 2. Since the MSE values are sufficiently small for both models, it can be concluded that the two models provided decent accuracy to move to the closed-loop simulation.

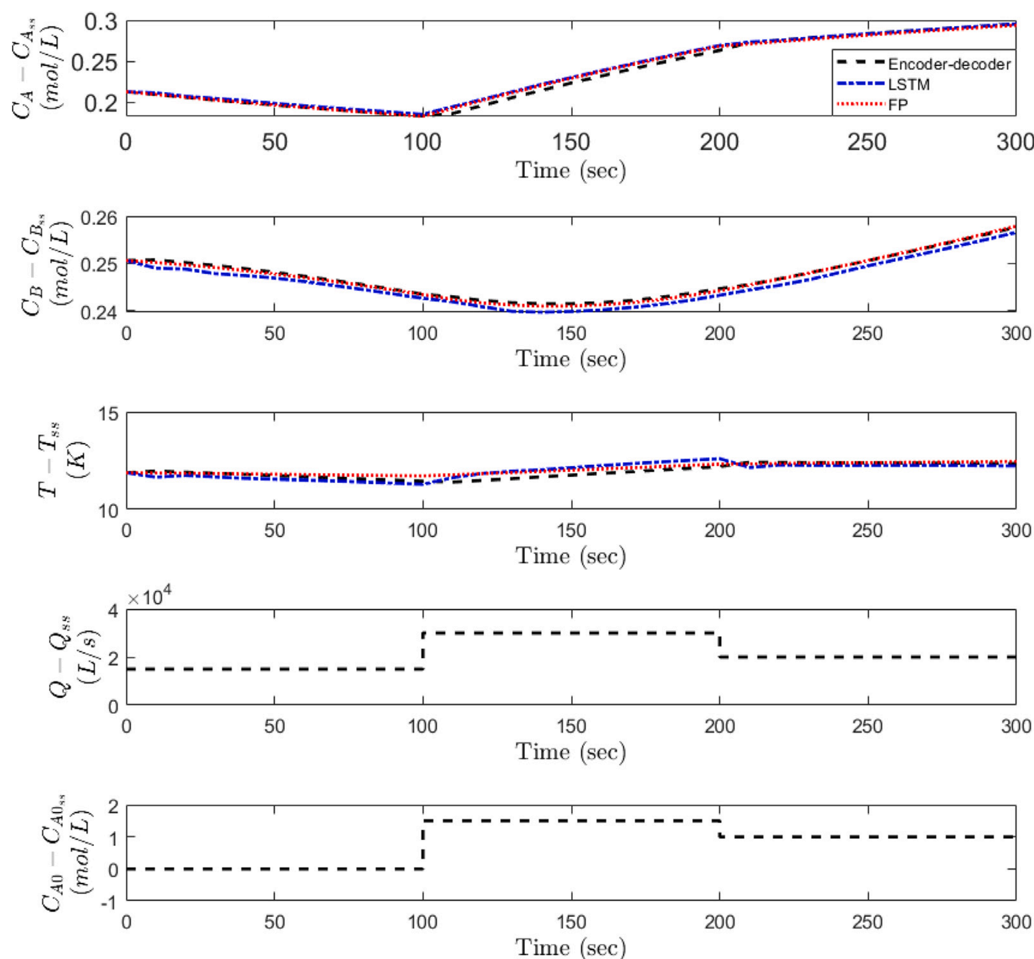


Fig. 7. Open-loop state and manipulated input profiles for the CSTR example.

Table 3
Final offset for each state from steady-state under LMPC using RNN and encoder–decoder models.

	$C_A - C_{A_{ss}}$ [mol/L]	$C_B - C_{B_{ss}}$ [mol/L]	$T - T_{ss}$ [K]
LSTM	3.7×10^{-3}	1.8×10^{-2}	7.3×10^{-2}
Encoder–decoder	7.8×10^{-5}	-1.2×10^{-4}	7.9×10^{-3}

6.5. Closed-loop performance

After the open-loop simulation, the models were tested in a closed-loop simulation of the underlying process under LMPCs based on the LSTM model and the encoder–decoder model. The dynamics of the closed-loop process under each controller is illustrated in Figs. 8 and 9. The states trajectories are shown in Fig. 8, while the associated manipulated inputs (i.e., control actions) are shown in Fig. 9. From the resulting trajectories, it can be seen that the LMPC based on the encoder–decoder model outperforms the RNN-based LMPC in terms of smoothness of the state profiles, but the controller was able to drive the three states towards steady-state and stabilize the system with both models. Furthermore, the encoder–decoder based LMPC was able to drive its states closer to the steady-state state values as shown in Table 3.

Remark 10. Different state observers, designed on the basis of neural network models, can be used to provide an estimate of the unmeasured states from the measured ones in the event that the state vector x is not entirely available online. In a previous work, Alhajeri et al. (2021) proposed two distinct machine learning-based state estimators for nonlinear processes within the framework of ML-based Lyapunov-based MPC. It was shown that all state trajectories converged to the steady-state under the LMPC starting from different initial conditions, and that both the fully ML-based and the hybrid-model-based state estimators provided accurate estimation.

6.6. MPC computation time considerations

One area not explored in the neural network-MPC example is the computational time necessary to find the optimal control action. In practice, there is a limit on the time allocated for the MPC to spend on solving the nonlinear optimization problem to ensure closed-loop stability. Some factors that affect the MPC's speed are the model's inference time, initial guesses, and optimization solver. Depending on what type of model is used in the MPC, the model's inference time can vary dramatically. A simple first-principles model takes less time than a traditional machine learning model, which takes less time than a deep neural network learning model. The inference time between different neural networks can also differ depending on their size and architecture. The inference time of the models used in this work is in the range of 0.1~0.5 s. In addition, the initial guess and the solver choice used in the optimization problem can also greatly affect the computational load. If the MPC optimization problem is to be solved with a poor initial guess, an inadequate solver, or if the predictive model takes too long to calculate, it is possible that the nonlinear optimization solver will not converge to a solution in the time allotted, resulting in a suboptimal MPC (Scokaert et al., 1999). The bottleneck in the MPC is heavily process-dependent and needs to be identified

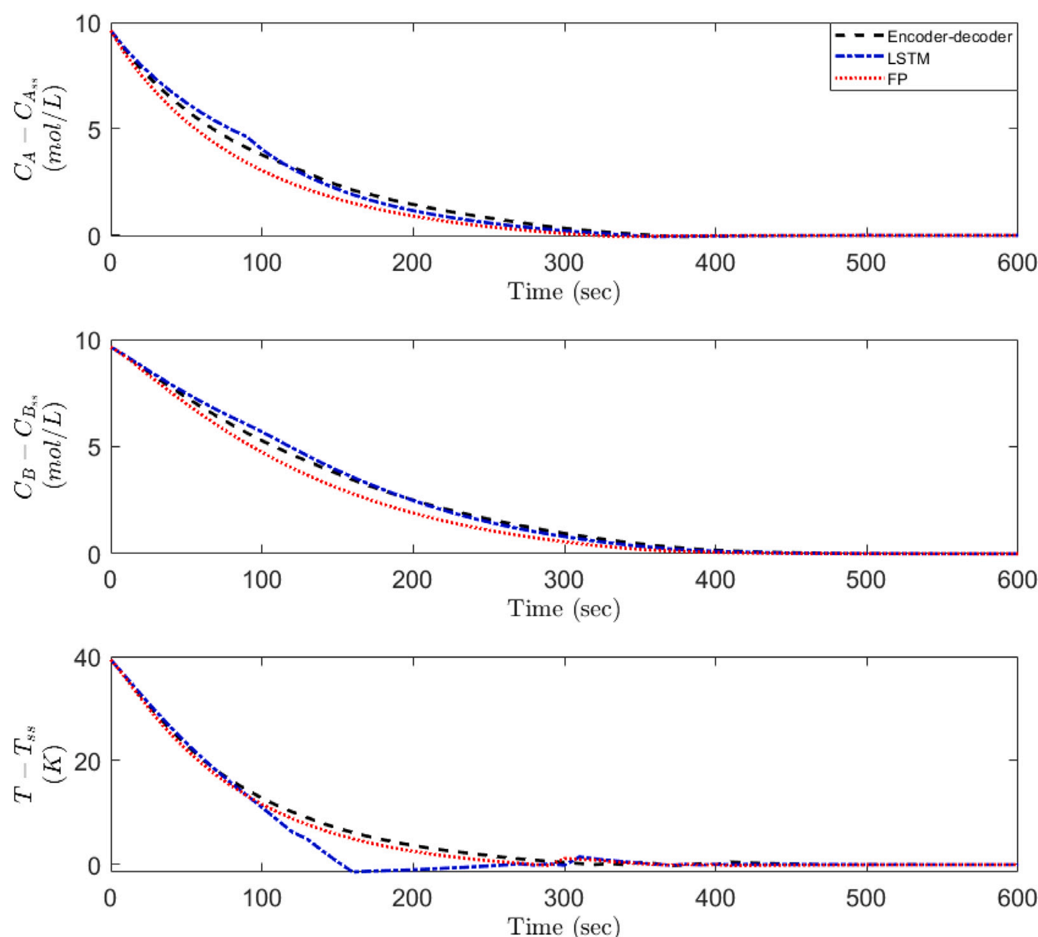


Fig. 8. State profiles of the closed-loop simulation of the first-principles process model under the LMPC using three models: first-principles (FP), LSTM, and encoder–decoder.

when attempting to improve the controller performance. For example, in a very complex process, the model required to capture the dynamics may be architecturally advanced and lead to a high model inference time, while a high-dimensional system will require a large number of values to be provided as initial guesses, which may be non-trivial and the bottleneck in solving the MPC optimization problem.

7. Conclusion and future directions

A survey on several neural network modeling approaches, in particular FNN, vanilla RNN, GRU, LSTM, and encoder–decoder architecture-based RNN, and their integration with MPC was discussed in this work. In addition, a tutorial was provided on the construction of the aforementioned neural network models with remarks on dealing with specific scenarios such as noisy data. Finally, a chemical process example was studied in closed-loop under the different neural network model-based MPCs to demonstrate the advantages and disadvantages of each model.

For future research directions, a recently proposed family of neural networks, named neural ordinary differential equations (ODE), provides the potential to improve the performance of modeling of continuous time-series data. This method proposes to parameterize an ODE between the states of a neural network and, as a result, the output of the neural network is the solution of an ODE initial value problem, which is computed with an explicit ODE solver (e.g., Euler method, Runge–Kutta methods) (Chen et al., 2018). Compared to traditional RNNs that are usually interpreted as discrete approximations of time-series data, theoretically, neural ODEs can be interpreted as continuous approximations of the data. In particular, recent applications of neural ODEs in the literature include improved performance for forecasting time-series data, especially for datasets with large or irregular sampling times (e.g., Rubanova et al., 2019). A potential future direction includes using neural ODEs as the process model for MPC to improve the performance of the MPC when dense and synchronous measurements are not available for model training purposes.

On the other hand, another neural network approach to consider is the transformer architecture. Up to this point, only sequential neural network models such as RNN, LSTM, GRU, and encoder–decoder systems were discussed. These sequential models generally have a recurrent structure which allows them to easily capture the ordinal aspect of time-series data. However, a drawback of the recurrent property is that they are not optimized for parallel computation. The input sequence to sequential models is provided the inputs one element at a time, which does not allow for parallel training and batch inference. In the example of an RNN, the hidden state of one RNN unit needs to be calculated first before being used as an input to the next RNN unit. In addition, sequential models assume that the previous recurrent unit is able to fully capture past behaviors as the current unit does not have direct access to past non-immediate units. Therefore, Vaswani et al. (2017) have developed a new deep learning architecture, called transformer, in order to overcome these limitations associated with sequential models. In the context of MPC, the implementation of transformers can potentially not only improve model accuracy but also speed up the MPC's computation time due to the faster model inference as a result of parallelization.

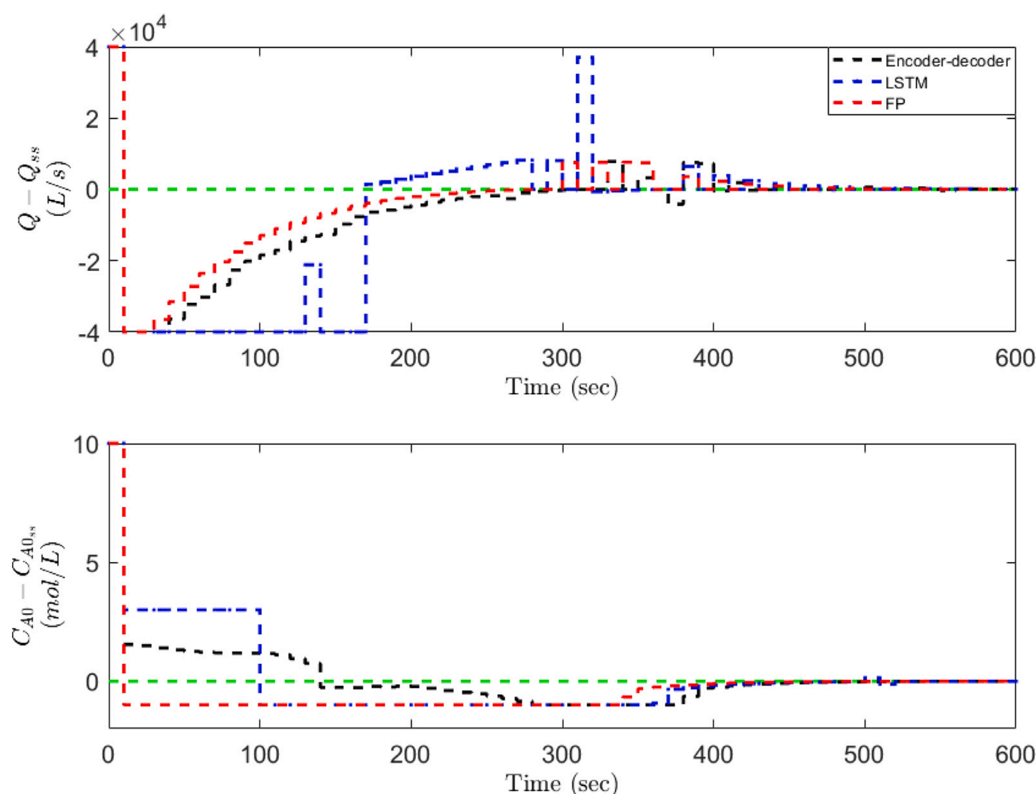


Fig. 9. Input profiles of the closed-loop simulation of the first-principles process model under the LMPC using three models: first-principles (FP), LSTM, and encoder–decoder.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

Financial support from the National Science Foundation and the Department of Energy is gratefully acknowledged.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al., 2016. TensorFlow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation. OSDI 16, Savannah, GA, USA, pp. 265–283.

Abdullah, F., Wu, Z., Christofides, P.D., 2021a. Data-based reduced-order modeling of nonlinear two-time-scale processes. Chem. Eng. Res. Des. 166, 1–9.

Abdullah, F., Wu, Z., Christofides, P.D., 2021b. Sparse-identification-based model predictive control of nonlinear two-time-scale processes. Comput. Chem. Eng. 153, 107411.

Abdullah, F., Wu, Z., Christofides, P.D., 2022. Handling noisy data in sparse model identification using subsampling and co-teaching. Comput. Chem. Eng. 157, 107628.

Afram, A., Janabi-Sharifi, F., Fung, A.S., Raahemifar, K., 2017. Artificial neural network (ANN) based model predictive control (MPC) and optimization of HVAC systems: A state of the art review and case study of a residential HVAC system. Energy Build. 141, 96–113.

Alanqar, A., Durand, H., Christofides, P.D., 2015a. On identification of well-conditioned nonlinear systems: Application to economic model predictive control of nonlinear processes. AIChE J. 61, 3353–3373.

Alanqar, A., Durand, H., Christofides, P.D., 2017. Error-triggered on-line model identification for model-based feedback control. AIChE J. 63 (3), 949–966.

Alanqar, A., Ellis, M., Christofides, P.D., 2015b. Economic model predictive control of nonlinear process systems using empirical models. AIChE J. 61, 816–830.

Alhajeri, M., Luo, J., Wu, Z., Albalawi, F., Christofides, P.D., 2022. Process structure-based recurrent neural network modeling for predictive control: A comparative study. Chem. Eng. Res. Des. 179, 77–89.

Alhajeri, M., Soroush, M., 2020. Tuning guidelines for model-predictive control. Ind. Eng. Chem. Res. 59 (10), 4177–4191.

Alhajeri, M.S., Wu, Z., Rincon, D., Albalawi, F., Christofides, P.D., 2021. Machine-learning-based state estimation and predictive control of nonlinear processes. Chem. Eng. Res. Des. 167, 268–280.

AlMomani, A.A.R., Sun, J., Bollt, E., 2020. How entropic regression beats the outliers problem in nonlinear system identification. Chaos 30, 013107.

Amrit, R., Rawlings, J.B., Angeli, D., 2011. Economic optimization using model predictive control with a terminal cost. Annu. Rev. Control 35 (2), 178–186.

Angeli, D., Amrit, R., Rawlings, J.B., 2011. On average performance and stability of economic model predictive control. IEEE Trans. Automat. Control 57 (7), 1615–1626.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13, 281–305.

Biegler, L.T., 2010. Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes. SIAM.

Billings, S.A., 1980. Identification of nonlinear systems-a survey. In: IEE Proceedings D-Control Theory and Applications, Vol. 127, no. 6. pp. 272–285.

Bonassi, F., Farina, M., Xie, J., Scattolini, R., 2022. On Recurrent Neural Networks for learning-based control: Recent results and ideas for future developments. J. Process Control 114, 92–104.

Brunton, S.L., Proctor, J.L., Kutz, J.N., 2016. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. Proc. Natl. Acad. Sci. 113, 3932–3937.

Camacho, E., Bordons, C., 2013. Model Predictive Control, second ed. Springer Science & Business Media, London.

Chang, H.-C., Aluko, M., 1984. Multi-scale analysis of exotic dynamics in surface catalyzed reactions I: Justification and preliminary model discriminations. Chem. Eng. Sci. 39 (1), 37–50.

Chen, R.T., Rubanova, Y., Bettencourt, J., Duvenaud, D.K., 2018. Neural ordinary differential equations. Adv. Neural Inf. Process. Syst. 31.

Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y., 2014a. On the properties of neural machine translation: Encoder-decoder approaches. In: Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation. Association for Computational Linguistics, Doha, Qatar, pp. 103–111.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014b. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP, Doha, Qatar, pp. 1724–1734.

Chow, T.W., Fang, Y., 1998. A recurrent neural-network-based real-time learning control strategy applying to nonlinear systems with unknown dynamics. IEEE Trans. Ind. Electron. 45 (1), 151–161.

Chow, G.C., et al., 1975. Analysis and Control of Dynamic Economic Systems. Wiley.

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

Clarke, D., Gawthrop, P., 1975. Self-tuning controller. Proc. Inst. Electr. Eng. 122, 929–934.

Clarke, D., Gawthrop, P., 1979. Self-tuning control. Proc. Inst. Electr. Eng. 126 (6), 633–640.

Csáji, B.C., et al., 2001. Approximation with artificial neural networks. Fac. Sci. Etvs Lornd Univ. Hungary 24 (48), 7.

Dietterich, T.G., 2002. Machine learning for sequential data: A review. In: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition. SSPR, Springer, pp. 15–30.

Domingos, P.M., 1997. Why does bagging work? A Bayesian account and its implications. KDD 155–158.

Donate, J.P., Cortez, P., Sanchez, G.G., De Miguel, A.S., 2013. Time series forecasting using a weighted cross-validation evolutionary artificial neural network ensemble. Neurocomputing 109, 27–32.

Draeger, A., Engell, S., Ranke, H., 1995. Model predictive control using neural networks. IEEE Control Syst. Mag. 15 (5), 61–66.

Ellis, M.J., Chinde, V., 2020. An encoder–decoder LSTM-based EMPC framework applied to a building HVAC system. Chem. Eng. Res. Des. 160, 508–520.

Ellis, M., Durand, H., Christofides, P.D., 2014. A tutorial review of economic model predictive control methods. J. Process Control 24 (8), 1156–1178.

Esche, E., Weigert, J., Rihm, G.B., Göbel, J., Repke, J.-U., 2022. Architectures for neural networks as surrogates for dynamic systems in chemical engineering. Chem. Eng. Res. Des. 177, 184–199.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. KDD 96 (34), 226–231.

Fablet, R., Ouala, S., Herzet, C., 2018. Bilinear residual neural network for the identification and forecasting of geophysical dynamics. In: Proceedings of the 26th European Signal Processing Conference. Rome, Italy, pp. 1477–1481.

Gal, Y., Ghahramani, Z., 2016. A theoretically grounded application of dropout in recurrent neural networks. Adv. Neural Inf. Process. Syst. 29.

González-García, R., Rico-Martínez, R., Kevrekidis, I., 1998. Identification of distributed parameter systems: A neural net based approach. Comput. Chem. Eng. 22, S965–s968.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

Gurney, K., 2018. An Introduction to Neural Networks. CRC Press.

Habib, U., Zucker, G., Blochle, M., Judex, F., Haase, J., 2015. Outliers detection method using clustering in buildings data. In: IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society. IEEE, pp. 000694–000700.

Han, B., Yao, Q., Yu, X., Niu, G., Xu, M., Hu, W., Tsang, I., Sugiyama, M., 2018. Co-teaching: Robust training of deep neural networks with extremely noisy labels. Adv. Neural Inf. Process. Syst. 31.

Hedjar, R., 2013. Adaptive neural network model predictive control. Int. J. Innovative Comput. Inf. Control 9 (3), 1245–1257.

Henson, M.A., Seborg, D.E., 1997. Nonlinear Process Control. Prentice Hall PTR Upper Saddle River, New Jersey.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Holkar, K., Waghmare, L., 2010. An overview of model predictive control. Int. J. Control Autom. 3 (4), 47–63.

Hopfield, J.J., 1982. Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. 79 (8), 2554–2558.

Hrovat, D., Di Cairano, S., Tseng, H., Kolmanovsky, I., 2012. The development of model predictive control in automotive industry: A survey. In: 2012 IEEE International Conference on Control Applications. Dubrovnik, Croatia, pp. 295–302.

Kieu, T., Yang, B., Jensen, C.S., 2018. Outlier detection for multidimensional time series using deep neural networks. In: 2018 19th IEEE International Conference on Mobile Data Management. MDM, Aalborg, Denmark, pp. 125–134.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kittisupakorn, P., Thitiyasook, P., Hussain, M.A., Daosud, W., 2009. Neural network based model predictive control for a steel pickling process. J. Process Control 19 (4), 579–590.

Kotsiantis, S.B., Kanellopoulos, D., Pintelas, P.E., 2006. Data preprocessing for supervised leaning. Int. J. Comput. Sci. 1 (2), 111–117.

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 521 (7553), 436–444.

Lee, J.H., 2000. Modeling and identification for NonlinearModel predictive control: Requirements, current status and future research needs. In: Nonlinear Model Predictive Control. Springer, pp. 269–293.

Lévine, J., Rouchon, P., 1991. Quality control of binary distillation columns via nonlinear aggregated models. Automatica 27 (3), 463–480.

Li, D., Chen, D., Jin, B., Shi, L., Goh, J., Ng, S.-K., 2019. MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks. In: International Conference on Artificial Neural Networks. Springer, Munich, Germany, pp. 703–716.

Li, Y., Tong, Z., 2021. Model predictive control strategy using encoder-decoder recurrent neural networks for smart control of thermal environment. J. Build. Eng. 42, 103017.

Lin, Y., Sontag, E.D., 1991. A universal formula for stabilization with bounded controls. Systems Control Lett. 16 (6), 393–397.

Liu, H., Shah, S., Jiang, W., 2004. On-line outlier detection and data cleaning. Comput. Chem. Eng. 28 (9), 1635–1647.

Lu, L., Jin, P., Karniadakis, G.E., 2019. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. arXiv preprint arXiv:1910.03193.

Maclaurin, D., Duvenaud, D., Adams, R., 2015. Gradient-based hyperparameter optimization through reversible learning. In: International Conference on Machine Learning. PMLR, Lille, France, pp. 2113–2122.

Maner, B.R., Doyle III, F.J., 1997. Polymerization reactor control using autoregressive-plus Volterra-based MPC. AIChE J. 43 (7), 1763–1784.

Mayne, D.Q., Rawlings, J.B., Rao, C.V., Scokaert, P.O.M., 2000. Constrained model predictive control: Stability and optimality. Automatica 36, 789–814.

Mendes-Moreira, J., Soares, C., Jorge, A.M., Sousa, J.F.D., 2012. Ensemble approaches for regression: A survey. ACM Comput. Surv. 45 (1), 1–40.

Miljanovic, M., 2012. Comparative analysis of recurrent and finite impulse response neural networks in time series prediction. Indian J. Comput. Sci. Eng. 3 (1), 180–191.

Miller, W.T., Sutton, R.S., Werbos, P.J., 1995. Neural Networks for Control. MIT Press.

Mohajerin, N., Waslander, S.L., 2019. Multistep prediction of dynamic systems with recurrent neural networks. IEEE Trans. Neural Netw. Learn. Syst. 30 (11), 3370–3383.

Mohanty, S., 2009. Artificial neural network based system identification and model predictive control of a flotation column. J. Process Control 19 (6), 991–999.

Morari, M., Lee, J.H., 1999. Model predictive control: Past, present and future. Comput. Chem. Eng. 23 (4), 667–682.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., 2019. Pytorch: An imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. 32.

Pemberton, J., 1990. Non-Linear and Non-Stationary Time Series Analysis, JSTOR.

Polikar, R., 2012. Ensemble learning. In: Ensemble Machine Learning. Springer, pp. 1–34.

Qin, S.J., Badgwell, T.A., 2003. A survey of industrial model predictive control technology. Control Eng. Pract. 11 (7), 733–764.

Qin, S.J., McAvoy, T.J., 1992. Nonlinear PLS modeling using neural networks. Comput. Chem. Eng. 16 (4), 379–391.

Raissi, M., Perdikaris, P., Karniadakis, G.E., 2018. Multistep neural networks for data-driven discovery of nonlinear dynamical systems. arXiv:1801.01236.

Rawlings, J.B., 2000. Tutorial overview of model predictive control. IEEE Control Syst. Mag. 20, 38–52.

Ribeiro, M.H.D.M., Stefenon, S.F., de Lima, J.D., Nied, A., Mariani, V.C., Coelho, L.d.S., 2020. Electricity price forecasting based on self-adaptive decomposition and heterogeneous ensemble learning. Energies 13 (19), 5190.

Rubanova, Y., Chen, R.T., Duvenaud, D.K., 2019. Latent ordinary differential equations for irregularly-sampled time series. Adv. Neural Inf. Process. Syst. 32.

Rudy, S.H., Nathan Kutz, J., Brunton, S.L., 2019. Deep learning of dynamics and signal-noise decomposition with time-stepping constraints. J. Comput. Phys. 396, 483–506.

Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. Nature 323 (6088), 533–536.

Schmidhuber, J., 2015. Deep learning in neural networks: An overview. Neural Netw. 61, 85–117.

Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 45 (11), 2673–2681.

Scokaert, P.O., Mayne, D.Q., Rawlings, J.B., 1999. Suboptimal model predictive control (feasibility implies stability). IEEE Trans. Automat. Control 44 (3), 648–654.

Snoek, J., Larochelle, H., Adams, R.P., 2012. Practical bayesian optimization of machine learning algorithms. Adv. Neural Inf. Process. Syst. 25.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. Adv. Neural Inf. Process. Syst. 27.

Tabuada, P., 2007. Event-triggered real-time scheduling of stabilizing control tasks. IEEE Trans. Automat. Control 52 (9), 1680–1685.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems. pp. 5998–6008.

Wang, X., Lemmon, M.D., 2008. Event design in event-triggered feedback control systems. In: 2008 47th IEEE Conference on Decision and Control. Cancun, Mexico, pp. 2105–2110.

Wellstead, P.E., Prager, D., Zanker, P., 1979. Pole assignment self-tuning regulator. In: Proceedings of the Institution of Electrical Engineers, Vol. 126, no. 8. IET, pp. 781–787.

Werbos, P.J., 1990. Backpropagation through time: What it does and how to do it. Proc. IEEE 78 (10), 1550–1560.

Wilson, Z.T., Sahinidis, N.V., 2017. The ALAMO approach to machine learning. Comput. Chem. Eng. 106, 785–795.

Wong, W.C., Chee, E., Li, J., Wang, X., 2018. Recurrent neural network-based model predictive control for continuous pharmaceutical manufacturing. Mathematics 6 (11), 242.

Wu, Z., Alnajdi, A., Gu, Q., Christofides, P.D., 2022. Statistical machine-learning-based predictive control of uncertain nonlinear processes. AIChE J. 68, e17642.

Wu, Z., Luo, J., Rincon, D., Christofides, P.D., 2021a. Machine learning-based predictive control using noisy data: Evaluating performance and robustness via a large-scale process simulator. Chem. Eng. Res. Des. 168, 275–287.

Wu, Z., Rincon, D., Christofides, P.D., 2019a. Real-time adaptive machine-learning-based predictive control of nonlinear processes. Ind. Eng. Chem. Res. 59 (6), 2275–2290.

Wu, Z., Rincon, D., Christofides, P.D., 2020. Process structure-based recurrent neural network modeling for model predictive control of nonlinear processes. J. Process Control 89, 74–84.

Wu, Z., Rincon, D., Gu, Q., Christofides, P.D., 2021b. Statistical machine learning in model predictive control of nonlinear processes. Mathematics 9 (16), 1912.

Wu, Z., Tran, A., Ren, Y.M., Barnes, C.S., Chen, S., Christofides, P.D., 2019b. Model predictive control of phthalic anhydride synthesis in a fixed-bed catalytic reactor via machine learning modeling. Chem. Eng. Res. Des. 145, 173–183.

Wu, Z., Tran, A., Rincon, D., Christofides, P.D., 2019c. Machine learning-based predictive control of nonlinear processes. Part I: Theory. AIChE J. 65 (11), e16729.

Xu, J., Li, C., He, X., Huang, T., 2016. Recurrent neural network for solving model predictive control problem in application of four-tank benchmark. Neurocomputing 190, 172–178.

Xu, Y., Xie, L., Dai, W., Zhang, X., Chen, X., Qi, G.-J., Xiong, H., Tian, Q., 2021. Partially-connected neural architecture search for reduced computational redundancy. IEEE Trans. Pattern Anal. Mach. Intell. 43 (9), 2953–2970.

Yin, S., Kaynak, O., 2015. Big data for modern industry: Challenges and trends [point of view]. Proc. IEEE 103 (2), 143–146.

Zarzycki, K., Ławryńczuk, M., 2021. LSTM and GRU neural networks as models of dynamical processes used in predictive control: A comparison of models developed for two chemical reactors. Sensors 21 (16), 5625.

Zefrehi, H.G., Altınçay, H., 2020. Imbalance learning using heterogeneous ensembles. Expert Syst. Appl. 142, 113005.

Zhang, A., Lipton, Z.C., Li, M., Smola, A.J., 2021a. Dive into deep learning. arXiv preprint arXiv:2106.11342.

Zhang, C., Qin, Y., Zhu, X., Zhang, J., Zhang, S., 2006. Clustering-based missing value imputation for data preprocessing. In: 2006 4th IEEE International Conference on Industrial Informatics. IEEE, pp. 1081–1086.

Zhang, Z., Wu, Z., Rincon, D., Christofides, P.D., 2019. Real-time optimization and control of nonlinear processes using machine learning. Mathematics 7 (10), 890.

Zhang, B., Zou, G., Qin, D., Lu, Y., Jin, Y., Wang, H., 2021b. A novel encoder-decoder model based on read-first LSTM for air pollutant prediction. Sci. Total Environ. 765, 144507.

Zheng, Y., Wang, X., Wu, Z., 2022. Machine learning modeling and predictive control of the batch crystallization process. Ind. Eng. Chem. Res. 61, 5578–5592.

Zou, H., Hastie, T., 2005. Regularization and variable selection via the elastic net. J. R. Stat. Soc. Ser. B Stat. Methodol. 67 (2), 301–320.