MACHINE LEARNING Science and Technology

PAPER

OPEN ACCESS

RECEIVED
30 June 2022

REVISED
5 October 2022

ACCEPTED FOR PUBLICATION
14 October 2022

PUBLISHED
11 November 2022

A comparative study of different machine learning methods for dissipative quantum dynamics

Luis E Herrera Rodríguez[1,2], Arif Ullah[3], Kennet J Rueda Espinosa[1,2], Pavlo O Dral[3,*] and Alexei A Kananenka[1,*]

[1] Department of Physics and Astronomy, University of Delaware, Newark, DE 19716, United States of America
[2] Departmento de Física, Universidad Nacional de Colombia, Bogotá, DC Colombia
[3] State Key Laboratory of Physical Chemistry of Solid Surfaces, Fujian Provincial Key Laboratory of Theoretical and Computational Chemistry, Department of Chemistry, and College of Chemistry and Chemical Engineering, Xiamen University, Xiamen 361005, People's Republic of China
[*] Authors to whom any correspondence should be addressed.

E-mail: dral@xmu.edu.cn and akanane@udel.edu

Keywords: machine learning, open quantum systems, kernel ridge regression, spin-boson, recurrent neural networks, LSTM, GRU, CNN

Abstract

It has been recently shown that supervised machine learning (ML) algorithms can accurately and efficiently predict long-time population dynamics of dissipative quantum systems given only short-time population dynamics. In the present article we benchmarked 22 ML models on their ability to predict long-time dynamics of a two-level quantum system linearly coupled to harmonic bath. The models include uni- and bidirectional recurrent, convolutional, and fully-connected feedforward artificial neural networks (ANNs) and kernel ridge regression (KRR) with linear and most commonly used nonlinear kernels. Our results suggest that KRR with nonlinear kernels can serve as inexpensive yet accurate way to simulate long-time dynamics in cases where the constant length of input trajectories is appropriate. Convolutional gated recurrent unit model is found to be the most efficient ANN model.

1. Introduction

Simulation of the dynamics of quantum dissipative systems [1–3] is one of the most challenging problems in physics and chemistry. Quantum dissipation arises from the coupling of a quantum system to a thermal bath which consists of an infinite number of degrees of freedom (DOF). This results in a time irreversible dynamics of the system.

A multitude of numerically exact methods has been developed for quantum dynamics simulations including the hierarchical equations of motion (HEOM) [4, 5], multi-configurational time-dependent Hartree [6, 7], quasi-adiabatic propagator path integral [8], time evolving density matrix using orthogonal polynomials algorithm [9], time-dependent density matrix renormalization group [10], time-dependent Davydov ansatz [11], diagrammatic quantum Monte Carlo [12, 13], tensor-train split-operator Fourier transform [14], and the stochastic equation of motion approach [15–19]. In practice, however, none of these methods can be used to simulate long-time dynamics of quantum systems coupled to realistic baths containing large number of, generally, anharmonic DOFs.

Methods based on Nakajima–Zwanzig generalized quantum master equation (GQME) [20–26] including transfer tensor method (TTM) [27–31], allow to obtain long-time dynamics of the reduced density matrix of a quantum system at a significantly lower computational cost compared to numerically exact methods, provided the GQME memory kernels are available. However, in general, it is difficult to numerically calculate the exact memory kernel for GQME. TTM further reduces the computational cost compared to the direct solution of GQME but it still requires an input set of dynamical maps to be generated by a numerically accurate method rendering such approaches infeasible for systems with many DOFs. Machine learning (ML) offers an alternative route to accurate, yet greatly accelerated quantum dynamics calculations with minimum input information required [32–35].

Predicting the future time-evolution of quantum mechanical observables based on past values can be formulated as time series forecasting problem. A time series is a set of data points recorded in consecutive time intervals. Forecasting is a challenging part of time series data analysis. Traditional approaches to forecasting of time series utilize moving averages. One such approach, Autoregressive Integrated Moving Average (ARIMA) is a linear regression-based method which, together with its variations (e.g. autoregressive integrated moving average with explanatory variable, ARIMAX and seasonal autoregressive integrated moving average, SARIMA) [36], have become the standard tools for various time series problems [37, 38]. These approaches perform reasonably well for short-term forecasting, but the performance of these methods deteriorates severely for long-term predictions. Additionally, such methods require assumptions about the underlying data that have to be incorporated into the model.

ML is artificial intelligence-based data analysis technique that is data-driven as opposed to traditional model-driven approaches. Perhaps the most widely known ML tool is a feedforward neural network (FFNN) which is an artificial neural network (ANN or simply NN) wherein connections between the nodes do not form a cycle [39, 40]. In this kind of ANN the information moves only in one direction from the input nodes, through the hidden nodes (if any), and to the output nodes without using any feedback from the past input data. As a result, the output of any layer does not affect the training process performed on the same layer. In practice, FFNNs are rarely used for sequential data. Instead, it is preferred to use ANNs specifically developed for this type of data such as recurrent neural networks (RNNs).

RNNs [40–43] in general, are designed with built-in gates that function to store the previous inputs and leverage sequential information. RNNs utilize a loop in the network to preserve some information and thus function like a memory. This gives such feedback-based models the ability to learn from past data. RNNs are known to outperform ARIMA-based models in the long-time forecasting problems [44–46].

RNNs have been successfully applied to time series classification [47–49] and forecasting across many domains including financial data predictions [50–53], speech [54–59] and handwriting recognition [60, 61], natural language processing [62, 63], machine translation [64–66], healthcare [67–70], traffic speed prediction [71–73], music [74, 75], video [76], meteorology [77], molecular drug discovery [78, 79], demand forecasting [80, 81], gaming [82], remote sensing [83, 84], computer code generation [85] and others.

There are several variations of RNN-based models differing in their capabilities to remember input data. The vanilla RNN [40, 86], long short-term memory (LSTM) [40, 42, 55, 87, 88], and gated recurrent unit (GRU) [40, 89] are the most commonly used types of RNNs. The vanilla RNN is the first model of recurrent ANNs that was introduced [86]. Its ability to learn long-term dependencies is limited due to vanishing and exploding gradient problems. LSTM-based models are extensions of RNNs with significantly improved performance on long-term predictions. LSTM uses a set of gates to learn what information is worth to remember. GRU is another gated variant of RNN model developed by Chung *et al* [90]. They showed that GRU outperforms LSTM in some tasks. Josefowicz *et al* [91] also reported the superior performance of the GRU-based models, but noticed that the performance of LSTM can nearly match that of the GRU if proper initialization is performed.

Bidirectional RNNs (BRNNs) [92] are another extension of the standard unidirectional RNNs in which two RNNs are applied to the input data. Firstly, an RNN is applied on the input sequence which is then followed by the application of RNN on the reversed input sequence. This typically improves the accuracy of the model [80, 93] at a cost of slower training times [46, 80]. Several variants of BRNNs exist differing in the type of the underlying RNN cell. Bidirectional LSTMs (BLSTMs) [57] and bidirectional GRUs (BGRUs) [68] have been applied to various tasks [52, 57, 62, 70, 80, 82]. In some applications [46, 73], such as speech recognition, the better performance of BLSTM compared to unidirectional LSTM has been reported [57] which was anticipated given the nature of the task (text parsing and prediction of next words in the input sentence). In general, however, it is not clear what problems benefit from the bi-directional training.

Convolutional neural networks (CNNs) [94] are a special kind of ANNs designed for processing data that has a known grid-like topology [40]. For example, time-series data can be thought of as a one-dimensional (1D) grid taking samples at regular time intervals. One-dimensional CNNs (1D CNNs) have achieved promising results in time-series classification tasks [49, 95–102] in many domains including healthcare [103–105], speech recognition [59, 106], music classification [107, 108], natural language processing [63] and others.

CNNs have also been combined with RNNs. Such hybrid models are called convolutional recurrent neural networks (CRNNs). A CRNN is a deep ANN that contains a CNN layer(s) followed by an RNN layer(s). Such architectures possess several advantages. Firstly, 1D CNN layers learn to sub-sample the data and reduce the input vector that is passed to an RNN layer(s). This is important because GRU and LSTM layers are computationally expensive and replacing at least some of them with convolutional layer(s) improves the computational scaling of an algorithm. Secondly, 1D CNN layers extract local information from neighboring time points and pass already detected temporal dependencies further down to RNN layers.

CRNNs are being actively explored in various tasks [104, 107–110]. It has been reported that CRNNs can outperform CNNs in some problems [107]. CNNs were combined with BLSTMs as well [111]. Note that these methods are different from convolutional LSTM models [112] in which input transformations and recurrent transformations are both convolutional.

Kernel methods represent another class of ML methods that are applied to time-series analysis [113–115]. Such methods employ a function (kernel) that maps input data into a high dimensional space and then perform a linear regression in that space. Kernel ridge regression (KRR) and support vector regression are examples of such algorithms [116]. A crucial aspect of applying kernel methods to time series data is to find appropriate kernels to distinguish between time series. A simple way is to treat time series as static vectors, essentially as they are treated in FFNNs, ignoring the time dependence. In such cases standard kernels such as Matérn [117] or Gaussian kernel function [116] can be used. However, such methods are limited to input time-sequences of equal length, again similar to FFNNs, yet many applications involve time sequences of varying length. To overcome this limitation, kernel functions such as autoregressive kernel have been developed [118].

Recently, many ML models have been applied to simulate the dynamics of quantum systems [32–35, 119–128]. We note that ML can also be applied to quantum dynamics in a different context—namely as surrogate models for quantum chemical properties such as potential energies and forces in different electronic states as well as couplings between the states eliminating the need for expensive (excited-state) electronic structure calculations [129–131]. Here we apply ML to propagate a quantum system assuming potential energies are readily available. ML models are attractive because of their very low computational cost and favorable scaling with respect to the size of a quantum system and bath. RNNs were used to simulate dynamics of the spin-boson model and Landau–Zener transitions [121, 122], and to learn the convolutionless master equation [123]. Rodríguez and Kananenka [32] used CNNs to accurately model long-time dynamics of the spin-boson system based on short-time dynamical data. Later Ullah and Dral [33] illustrated that KRR methods can also predict long-time dynamics of the spin-boson model very accurately. Recently, CNNs were used to study the excitation energy transfer in Fenna–Matthews–Olson light-harvesting complex [34, 35]. Wu *et al* [124] used a hybrid CNN/LSTM network to predict long-time semiclassical and mixed quantum–classical dynamics of the spin-boson model. Lin *et al* [125, 132] trained a multi-layer LSTM model to simulate the long-time dynamics of spin-boson model and used the bootstrap method to estimate the confidence interval. In a recent study, the same authors used LSTM to predict semiclassical dynamics based on symmetrical quasi-classical method [133]. We note that ML methods based on FFNNs have also been recently applied to model quantum dynamics [119, 120].

The recent upsurge of applications of ML methods to dissipative quantum dynamics calls for a systematic benchmark of such methods. In this article, we present a comprehensive comparison of 22 ML models for predicting the long-time dynamics of an open quantum system given the short-time evolution data. We consider all three most used types of unidirectional RNNs including the vanilla RNN, GRU, and LSTM, the corresponding BRNNs (BRNN, BGRU, BLSTM), 1D CNN, CRNN, convolutional BRNNs (CBRNNs), as well as KRR with eight different kernel functions. We test the ability of these ML methods to predict the donor-acceptor population difference of a spin-boson model in symmetric and asymmetric regimes over a broad range of temperatures, reorganization energies, and bath relaxation timescales.

2. Theory

2.1. Model system

We choose to test ML methods on the spin-boson model which has become a paradigmatic model system in the study of open quantum systems due to the richness of its physics [3]. The spin-boson model was exploited in a wide-range of applications in quantum computing [134], quantum phase transitions [135, 136], electron transfer in biological systems [137], and others. The spin-boson model describes a two-level quantum system linearly coupled to a heat-bath environment. The bath is modeled as an ensemble of independent harmonic oscillators. The total Hamiltonian in the system's basis $\{|+\rangle, |-\rangle\}$ is given by $(\hbar = 1)$

$$\hat{H} = \frac{\epsilon}{2}\hat{\sigma}_z + \frac{\Delta}{2}\hat{\sigma}_x + \hat{\sigma}_z \sum_\alpha g_\alpha \left(b_\alpha^\dagger + b_\alpha\right) + \sum_\alpha \omega_\alpha b_\alpha^\dagger b_\alpha, \tag{1}$$

where $b_\alpha^\dagger$ (b_α) is the bosonic creation (annihilation) operator of the αth mode with frequency ω_α, $\hat{\sigma}_z = |+\rangle\langle+| - |-\rangle\langle-|$ and $\hat{\sigma}_x = |+\rangle\langle-| + |-\rangle\langle+|$ are the Pauli operators, ϵ is the energetic bias, Δ is the tunneling matrix element, and g_α are the coupling coefficients.

The impact of the bath is completely determined by the spectral density $J(\omega) = \pi \sum_\alpha g_\alpha^2\, \delta(\omega_\alpha - \omega)$ which, in this work, is chosen to be of the Debye form (Ohmic spectral density with the Drude–Lorentz cut-off) [138]

$$J(\omega) = 2\lambda \frac{\omega\omega_c}{\omega^2 + \omega_c^2}, \tag{2}$$

where λ is the bath reorganization energy which controls the strength of system-bath coupling, and the cutoff frequency ω_c which sets the primary timescale for the bath evolution $\tau_c = (\omega_c)^{-1}$.

We consider the time evolution of the expectation value of $\hat{\sigma}_z$ Pauli operator

$$\langle \hat{\sigma}_z(t) \rangle = \mathrm{Tr}_\mathrm{s}\left[\hat{\sigma}_z \hat{\rho}_\mathrm{s}(t)\right], \tag{3}$$

which is often referred to as the population difference $\langle \hat{\sigma}_z(t) \rangle = p_+(t) - p_-(t)$, where $p_\pm(t) = \mathrm{Tr}_\mathrm{s}$ $[|\pm\rangle\langle\pm|\hat{\rho}_\mathrm{s}(t)]$. In equation (3) the trace is taken over the system DOFs as denoted by ‘s’, and $\hat{\rho}_\mathrm{s}$ is the system’s reduced density operator

$$\hat{\rho}_\mathrm{s}(t) = \mathrm{Tr}_\mathrm{b}\left[e^{-i\hat{H}t} \hat{\rho}(0) e^{i\hat{H}t}\right], \tag{4}$$

where $\hat{\rho}(0)$ is the total system plus bath density operator and the trace is taken over the bath DOFs. The initial state of the total system is assumed to be a product state of the following form

$$\hat{\rho}(0) = \hat{\rho}_\mathrm{s}(0) \otimes \frac{e^{-\beta \hat{H}_\mathrm{b}}}{Z_\mathrm{b}}, \tag{5}$$

where $\hat{H}_b = \sum_\alpha \omega_\alpha b_\alpha^\dagger b_\alpha$ is the bath Hamiltonian, $Z_\mathrm{b} = \mathrm{Tr}_\mathrm{b}[e^{-\beta \hat{H}_\mathrm{b}}]$ is the bath partition function, $\beta = (k_\mathrm{B} T)^{-1}$ is the inverse temperature, and k_B is the Boltzmann constant. The initial density operator of the system is chosen to be $\hat{\rho}_\mathrm{s}(0) = |+\rangle\langle+|$. These conditions correspond to situations where the initial preparation of the system occurs quickly on the timescale of the bath relaxation.

2.2. ML models

In this section we provide a detailed description of all ML models used in the present study. We specialize our discussion to modeling time-series data with the input sequence denoted as $\mathbf{x} = \left(x^{(1)}, \ldots, x^{(T)}\right)$ where T is the length of the time series. Each element $x^{(t)}$ of $\mathbf{x}$ can be a real-valued vector itself, $x^{(t)} \in \mathbb{R}^n$. Consider a data set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ containing N time series $\mathbf{x}_i$ and their associated labels $\mathbf{y}_i$. In time-series forecasting problems, labels can describe the future states of the input sequence $\mathbf{x}_i$ as denoted by $\mathbf{y}_i = \left(x_i^{(T+1)}, \ldots, x_i^{(T+m)}\right)$ which corresponds to m next elements of the sequence $\mathbf{x}_i$. Time-series forecasting in a supervised learning framework amounts to (i) training ML models on the subset of $\mathcal{D}$ called a training set and (ii) using trained ML models to make a prediction $\hat{\mathbf{y}}_i$ for a given test time series $\mathbf{x}_i$. In this work we test the ability of ML models to predict a single real-valued scalar quantity, the population difference of the spin-boson model $\langle \hat{\sigma}_z(t) \rangle$ for a single time step i.e. $n = 1$ and $m = 1$. Extension to multivariate time series data ($n > 1$) and multi-step outputs ($m > 1$) is possible [35].

2.2.1. FFNNs

FFNNs or multiplayer perceptrons are the most used type of ANNs. FFNNs approximate a function $g(\mathbf{x})$ by defining a mapping $\hat{\mathbf{y}} = \mathcal{F}(\mathbf{x}; \boldsymbol{\theta})$ and determining the value of parameters $\boldsymbol{\theta}$ that best approximate $g(\mathbf{x})$. These models are called feedforward because information flows through the function being evaluated from an input $\mathbf{x}$ through intermediate steps to the output $\hat{\mathbf{y}}$. There are no feedback connections between the output and the input of the model [40].

A crucial element of the success of ANNs is the use of deep architectures. Deep ANNs are created by stacking multiple layers on top of each other, with the output of one layer forming the input for the next layer. The first layer is called the input layer, the last layer is called the output layer; all layers in between are called hidden layers.

Deep FFNNs are compositions of several functions: $\hat{\mathbf{y}} = \mathcal{F}^{(L)}(\ldots \mathcal{F}^{(2)}(\mathcal{F}^{(1)}(\mathbf{x}; \boldsymbol{\theta}^{(1)})); \boldsymbol{\theta}^{(2)}); \boldsymbol{\theta}^{(L)})$ where each function $\mathcal{F}^{(l)}$ depends on its own set of parameters $\boldsymbol{\theta}^{(l)}$. Function $\mathcal{F}^{(l)}$ connects lth and $(l+1)$th layers of the network. It takes an input $\mathbf{x}^{(l)} \in \mathbb{R}^{k_l}$ and generates the output according to

$$\mathbf{x}^{(l+1)} = \mathcal{F}^{(l)}\left(\mathbf{x}^{(l)}; \boldsymbol{\theta}^{(l)}\right) = f^{(l)}\left(\mathbf{a}^{(l)}\right), \tag{6}$$

where $f^{(l)}: \mathbb{R} \to \mathbb{R}$ is the lth layer activation function which is applied elementwise. As seen from equation (6) each layer of the network is vector valued. Each vector element $a_j^{(l)}$ is called a neuron. Its value is calculated from layer’s input and parameters

$$a_j^{(l)} = \sum_{i=1}^{k_l} w_{ij}^{(l)} x_i^{(l)} + b_j^{(l)}, \tag{7}$$

where $\mathbf{w}^{(l)} \in \mathbb{R}^{k_l \times k_{l+1}}$ and $\mathbf{b}^{(l)} \in \mathbb{R}^{k_l}$ are called the weights and the biases of the lth layer, respectively, that together constitute the set of trainable parameters of the lth layer $\boldsymbol{\theta}^{(l)} = \{\mathbf{w}^{(l)}, \mathbf{b}^{(l)}\}$. The output $\mathbf{x}^{(l+1)}$ forms an input into $(l+1)$th layer. The total number of trainable parameters, biases and weights, of a single FFNN layer is $k_l(k_{l+1}+1)$. It should be emphasized that FFNN models require input time sequences of a fixed length, T. This, in particular, requires an *a priori* knowledge of the memory effects in a quantum system under study which is non-trivial task. ANNs based on recurrent layers, discussed below, do not impose such a restriction.

The strategy of deep learning is to find the set of model parameters $\{\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(L)}\}$ that best approximates the target function $g(\mathbf{x})$. The model parameters are adjusted during the training process which is based on backpropagation algorithm [40].

Typically the activation function of the hidden layers is chosen to be nonlinear, e.g. logistic sigmoid function $\sigma(z) = (1+e^{-z})^{-1}$. According to the universal approximation theorem [139–141] an FFNN with a linear output activation function and at least one hidden layer with a nonlinear activation function can approximate any Borel measurable function (continuous on the closed and bonded subset of the real coordinate space) from one finite-dimensional space to another with any desired nonzero amount of error, provided the network contains enough neurons. The theorem guarantees that regardless of the target function, a single hidden layer FFNNs with many neurons will be able to represent this function with any degree of accuracy. However, in general, there is no guarantee that the training algorithm can do so [40]. In practice, increasing neural network depth (including more hidden layers) often allows for more flexibility, although too deep neural networks may be difficult to train and easy to overfit. Thus, the number of hidden layers, their size and type (activation function), is a subject of optimization [40].

2.2.2. 1D CNNs

Convolutional neural networks (CNNs) is a type of ANNs that can be applied to time-series data [40, 94]. CNNs are based on a mathematical operation called convolution. Let $(\mathbf{g} * \mathbf{w})$ be the result of a 1D discrete convolution and the ith element of the result is given by

$$(\mathbf{g} * \mathbf{w})_i = \sum_{j=1}^{k} g_{i+j-1} w_{k+1-j}, \tag{8}$$

where $\mathbf{w} \in \mathbb{R}^k$ is referred to as kernel, or weight vector, and $\mathbf{g}$ is a time series. CNNs exploit two key ideas: sparse connectivity and parameter sharing [40]. The former is accomplished by making the kernel size smaller than the size of the input which allows to detect, for the time-series data, short-time correlations and reduces the number of parameters to be stored in memory compared to FFNNs. The latter amounts to using the same kernel parameters for all positions in the input which further reduces the storage requirements. In the case of convolution, the parameter sharing leads to equivariance—a property that causes the output to change in the same way the input changes. Specifically, in the case of time-series data, the convolution captures a timeline for different features to appear in the input [40].

In practical implementations of CNNs, the convolution operation given in equation (8) is often replaced by the so-called cross-correlation operation [40] which we denote by a '$\star$'

$$(\mathbf{g} \star \mathbf{w})_i = \sum_{j=1}^{k} g_{i+j-1} w_j. \tag{9}$$

It should be noted that equation (9) is given for the step size of the kernel, as it is applied across the sequence, equal to 1. The step size is called stride $s (s \in \mathbb{Z}^+)$. In general, for the stride $s \geqslant 1$, equation (9) is modified to

$$(\mathbf{g} \star \mathbf{w})_i = \sum_{j=1}^{k} g_{(i-1)s+j} w_j. \tag{10}$$

Let's consider a two-layer 1D CNN architecture. An input sequence $\mathbf{x}$ is processed through K_1 1D kernels of the first layer whose weights are denoted as $\mathbf{w}^{(1)} \in \mathbb{R}^{k_1}$. The result of this operation is tensor $\mathbf{Z}^{(1)}$ whose elements are given by

$$Z_{ij}^{(1)} = f^{(1)} \left[\left(\mathbf{x} \star \mathbf{w}_j^{(1)} \right)_i + b_j^{(1)} \right], \tag{11}$$

where $f^{(1)}$ is the activation function, $\mathbf{b}^{(1)} \in \mathbb{R}^{K_1}$ are the bias parameters and $\mathbf{w}_j^{(1)}, j = 1, \ldots, K_1$ is the jth kernel of the first layer. The output tensor $\mathbf{Z}^{(1)}$ has dimensions of $M_1 \times K_1$ with $M_1 = \lfloor (T - k_1 + 2p_1)/s_1 \rfloor + 1$, where s_1 is the stride, p_1 is the amount of zero padding, and $\lfloor x \rfloor = \max\{m \in \mathbb{Z} | m \leqslant x\}$ is the floor function. Zero padding parameter p_1 denotes the number of zeros to be added to the start and the end of the input sequence. For example, for $p = 0$ the input sequence is not padded with zeros and, consequently, the kernel is allowed to visit only positions where it is contained entirely within the input sequence. This type of zero padding is called valid padding.

Because convolution of input data with a single kernel can extract only one kind of features, albeit at many locations, in practice, many filters are used. Each element of a kernel is a trainable parameter and the same parameters of each kernel are shared across the entire sequence. Therefore, the total number of trainable parameters of the first 1D CNN layer is $K_1(k_1 + 1)$.

The output of the first layer $\mathbf{Z}^{(1)}$ is then processed through $K_1 \times K_2$ 1D kernels whose weights are denoted as $\mathbf{w}^{(2)} \in \mathbb{R}^{k_2}$. The output of the second layer is tensor $\mathbf{Z}^{(2)}$ whose elements are given by

$$Z_{ij}^{(2)} = f^{(2)} \left[\sum_{u=1}^{K_1} \left(\mathbf{Z}_u^{(1)} \star \mathbf{w}_{uj}^{(2)} \right)_i + b_j^{(2)} \right], \tag{12}$$

where $f^{(2)}$ is the activation function of the second layer and $\mathbf{b}^{(2)}$ are the bias parameters of the second layer. In equation (12) $\mathbf{w}_{uj}^{(2)}$ is to be understood as the uth kernel (weight vector) applied to the output of jth kernel (also called channel) of the first layer. The dimensions of $\mathbf{Z}^{(2)}$ are $M_2 \times K_2$ where $M_2 = \lfloor (M_1 - k_2 + 2p_2)/s_2 \rfloor + 1$ with p_2 and s_2 being the zero-padding and the stride of the second 1D CNN layer, respectively. It is worth emphasizing that a separate set of kernels is applied to the output of each kernel of the preceding (in this case first) layer. Thus, the total number of trainable parameters of the second 1D CNN layer is $K_2(K_1 k_2 + 1)$.

A pooling layer is commonly found in CNN architectures. Its function is to subsample (shrink) the input. Pooling function replaces the output of a CNN layer at a certain location with a neighborhood-dependent information. For example, the maximum pooling (MaxPooling) operation [142] outputs the maximum value in a neighborhood of a specified size. This helps to make representation approximately invariant to small translations of the input and improves the computational efficiency in a deep CNN by reducing the number of trainable parameters of the next convolutional layer. A 1D MaxPooling layer with the kernel size k_p, stride s_p, operates independently on every depth slice of the input $\mathbf{Z}^{\text{in}}$ and resizes it by taking only the maximum value

$$Z_{ij}^{\text{out}} = \max \left(Z_{(i-1)s_p+1,j}^{\text{in}}, \ldots, Z_{(i-1)s_p+k_p,j}^{\text{in}} \right), \tag{13}$$

where $i = 1, \ldots, \lfloor (M_2 - k_p)/s_p \rfloor + 1$. Note that equation (13) is written, for simplicity, for the zero padding but other kinds of padding can be used as well. MaxPooling layers do not employ any trainable parameters.

2.2.3. RNNs

RNNs are a family of neural networks specifically designed for processing sequential data [143]. Similarly to FFNNs, RNNs have the universal approximation ability [139, 144]. Similarly to CNNs, RNNs are based on the idea of parameter sharing but, unlike CNNs, RNNs share parameters through recursion. Formally, given a sequence $\mathbf{x}$ at each time step, an RNN updates its hidden state $\mathbf{H} = \left[\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(T)}\right]$ recursively based on the current input $x^{(j)}$ and the previous hidden state $\mathbf{h}^{(j-1)}$ as follows

$$\mathbf{h}^{(j)} = \mathcal{R}(\mathbf{h}^{(j-1)}, x^{(j)}), \tag{14}$$

where $\mathcal{R}$ is a nonlinear function. Each element of the hidden state vector, $\mathbf{h}^{(j)}$ is, in general, a vector itself. Recursive application of equation (14) results in the sharing of parameters across an ANN architecture. Training of such an ANN amounts to selectively emphasizing some aspects of the past sequence inputs that deemed more important than others.

All RNN models used in this work contain two stacked recurrent layers and are schematically illustrated in figure 1(a). The number of RNN cells in the first layer equals to the length of an input vector such that each cell processes a single time step $t = 1, \ldots, T$ and updates the corresponding hidden state vector $\mathbf{h}^{(t)} \in \mathbb{R}^{k_1}$. The size of the hidden state vector k_1 is a hyperparameter that needs to be optimized. Note that various software packages use different terminology. In this work RNN models are built with Keras [145] software whose argument units corresponds to the size of the hidden state vector. An RNN layer outputs a tensor, $\mathbf{H} \in \mathbb{R}^{T \times k_1}$, containing all hidden states and it serves as an input to the following RNN layer.

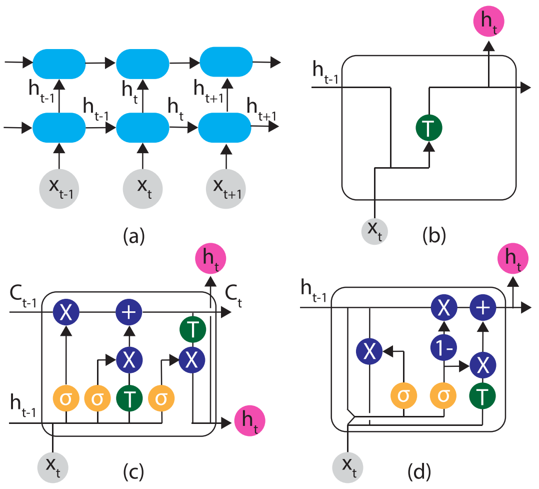

Figure 1. (a) Basic structure of a two-layer unidirectional recurrent neural network. Blue rounded rectangles represent cells and x_t is the tth element of an input sequence $\mathbf{x}$. Hidden state vector $\mathbf{h}^{(t)}$ is passed between the cells within a layer as well as between the layers; (b) the vanilla RNN cell with tanh activation function; (c) LSTM cell; (d) GRU cell. Green and yellow circles denote hyperbolic tangent and sigmoid activation functions correspondingly. Dark blue circles denote element-wise arithmetic operations shown on the circle. See text for details.

Each RNN cell of a layer processes the corresponding slice of the preceding layer's output $\mathbf{H}_{Lj}$, where $j \in [1, k_1]$ and L is the number of cells of the second layer, and transforms it into the state vector $\mathbf{h}_2^{(t)} \in \mathbb{R}^{k_2}$, where k_2 is the number of units of the second RNN layer. The last RNN layer can either return the hidden state vector at only the final time step T or the entire sequence of hidden states.

Various RNN architectures differ by the function $\mathcal{R}$ in equation (14). A simple RNN cell, also known as vanilla RNN is illustrated in figure 1(b) with the hyperbolic tangent activation function. It receives an input $x^{(t)}$ corresponding to the time step t and uses the state vector from the chronologically previous RNN cell $\mathbf{h}^{(t-1)} \in \mathbb{R}^k$ to generate the hidden state $\mathbf{h}^{(t)}$ according to the following update equation

$$\mathbf{h}^{(t)} = \tanh\left(\mathbf{b} + \mathbf{w}_h \cdot \mathbf{h}^{(t-1)} + \mathbf{w}_x \cdot \mathbf{x}^{(t)}\right), \qquad (15)$$

where $\mathbf{b}_h \in \mathbb{R}^k$ are the bias parameters, $\mathbf{w}_h \in \mathbb{R}^{k\times k}$, $\mathbf{w}_x \in \mathbb{R}^{k\times m}$ are the coefficient matrices, and the tanh activation function is applied element-wise. The function $\mathcal{R}(\mathbf{h}^{(j-1)}, x^{(j)})$ in case of the vanilla RNN amounts to iterative application of equation (15) to the input sequence $\mathbf{x}$ from $t = 1$ to $t = T$.

The hidden state $\mathbf{h}^{(t)}$ is then passed to the next RNN cell. Weights $\mathbf{w}_h, \mathbf{w}_x$ and biases $\mathbf{b}$ in equation (15) are updated iteratively via the back-propagation algorithm. In a multilayer RNN architecture all hidden state vectors $\mathbf{H}$ are passed to the next layer. For RNNs whose output is used directly to predict the next value of a sequence the hidden state corresponding to $t = T$ is the desired predicted value of the next time step $\hat{\mathbf{y}} = \mathbf{h}^{(T)}$.

The total number of trainable parameters of the vanilla RNN cell is $k(k+n+1)$, where k is the size of the hidden state vector, which is the same for all RNN cells within a layer, and n is the dimension of each element of an input sequence $\mathbf{x}^{(t)}$ which, in this work is 1 for the first RNN layer and k_1 for the second RNN layer. Note that if RNN is not the first layer, as e.g. in convolutional recurrent neural networks discussed below, n, for the first RNN layer, would be different from 1. It should also be noted that the total number of trainable parameters of the whole RNN layer does not depend on the number of RNN cells because $\mathbf{b}, \mathbf{w}_h$, and $\mathbf{w}_x$ are shared across all RNN cells within the given layer. The total number of trainable parameters in a two-layer vanilla RNN architecture is $k_2(k_1 + k_2 + 1) + k_1(k_1 + 1 + 1)$.

In spite of being simple and powerful ANN model, the vanilla RNN described above is plagued by the vanishing gradient and exploding gradient problems when trained via backpropagation [40, 146–148]. The exploding gradients problem refers to the large increase in the norm of the gradient during training. The vanishing gradients problem refers to the opposite behavior, when long term components decay exponentially fast to zero norm, limiting the model's ability to learn long-range dependencies. The exploding gradient problem can be easily addressed by gradient clipping [149]. The vanishing gradient problem is, however, much more difficult to address.

2.2.4. LSTM

In general, learning long-term dependencies is one of the most important problems in deep learning [40]. The difficulty in dealing with long-memory sequences in RNNs arises from the exponentially smaller weights given to long-term correlations compared to the short-term ones [40, 146–148]. This problem is particular to simple RNN cell described above. It was shown that very deep FFNNs can avoid the vanishing and exploding gradient problems [150] but, in order to store memories, RNNs must enter a region of parameter space where gradients vanish [147, 148].

To overcome the vanishing gradient problem various gated RNN architectures such as LSTM [42, 55, 87, 88] were developed. Gated RNNs are based on the idea of creating paths through time such that the derivatives are neither vanish nor explode. LSTM model was developed by Hochreiter *et al* [42] and is based on the idea of using self-loops to produce paths where the gradient can flow for long duration. This is achieved by adding the cell state denoted by $\mathbf{C}^{(t)} \in \mathbb{R}^k$ and data-dependent gates, that control the flow of information, to the standard RNN architecture [42, 87]. Both hidden state and cell state control the memory of the network. The cell state carries relevant information throughout the processing of the sequence such that even information from the earlier time steps can make its way to later time steps, reducing the effects of short-term memory. The information is added or removed to the cell state via gates. The gates are simple FFNNs that decide which information is allowed on the cell state. Thus, the gates can learn what information is relevant to keep or forget during training.

Although a number of variants of the LSTM cell have been produced, a large-scale analysis shows that none of them outperforms the standard LSTM architecture [42, 55, 87]. The block diagram of the LSTM cell is shown in figure 1(c). It contains three gates: the forget gate, the input gate, and the output gate. All gates have a sigmoid [logistic $\sigma(z) = (1+e^{-z})^{-1}$], nonlinear activation function (yellow circles). The tth element(s) of the input sequence $\mathbf{x}^{(t)}$ enters the cell and is concatenated with the hidden state vector from the chronologically previous cell $\mathbf{h}^{(t-1)}$. The total vector is passed through the forget gate

$$\mathbf{F}^{(t)} = \sigma\left(\mathbf{b}^{\mathrm{F}} + \mathbf{W}_h^{\mathrm{F}} \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_x^{\mathrm{F}} \cdot \mathbf{x}^{(t)}\right), \tag{16}$$

where $\mathbf{b}^{\mathrm{F}} \in \mathbb{R}^k$ are the biases and $\mathbf{W}_h^{\mathrm{F}} \in \mathbb{R}^{k\times k}$, $\mathbf{W}_x^{\mathrm{F}} \in \mathbb{R}^{k\times m}$, are the corresponding weights. The forget gate was the crucial addition to the original LSTM cell extending the length of sequences that can be processed [87]. The forget gate learns to reset memory blocks once their contents are out of date. The output vector $\mathbf{F}^{(t)} \in \mathbb{R}^k$ contains the values between 1 and 0 emphasizing or diminishing the importance of the elements of the hidden state vector, correspondingly. In two separate branches, the current input vector and the previous hidden state vector are processed through the input gates: the external input gate

$$\mathbf{I}^{(t)} = \sigma\left(\mathbf{b}^{\mathrm{I}} + \mathbf{W}_h^{\mathrm{I}} \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_x^{\mathrm{I}} \cdot \mathbf{x}^{(t)}\right), \tag{17}$$

and the new candidate gate

$$\mathbf{G}^{(t)} = \tanh\left(\mathbf{b}^{G} + \mathbf{W}_h^{G} \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_x^{G} \cdot \mathbf{x}^{(t)}\right), \tag{18}$$

where $\mathbf{b}^{\mathrm{I}} \in \mathbb{R}^k$, $\mathbf{b}^G \in \mathbb{R}^k$, $\mathbf{W}_h^I \in \mathbb{R}^{k\times k}$, $\mathbf{W}_h^G \in \mathbb{R}^{k\times k}$, $\mathbf{W}_x^I \in \mathbb{R}^{k\times m}$, and $\mathbf{W}_x^G \in \mathbb{R}^{k\times m}$ are the corresponding biases and weights of the input and new candidate gates. The external output gate uses the sigmoid activation function to obtain a gating value between 0 and 1 which is then element-wise multiplied by the corresponding element of the new candidate vector $\mathbf{G}^{(t)}$ effectively deciding which elements of the input vector are the most important. Given $\mathbf{F}^{(t)}$, $\mathbf{I}^{(t)}$, $\mathbf{G}^{(t)}$, and the cell state of the previous time step $\mathbf{C}^{(t-1)}$, $\mathbf{C}^{(t)}$ is updated as follows

$$\mathbf{C}^{(t)} = \mathbf{F}^{(t)} \odot \mathbf{C}^{(t-1)} + \mathbf{I}^{(t)} \odot \mathbf{G}^{(t)}, \tag{19}$$

where $\odot$ denotes the element-wise vector (Hadamard) product. The first term in equation (19) removes parts of the previous cell state through the forget gate and the second term adds new information yielding the new cell state.

Finally, the hidden state $\mathbf{h}^{(t)}$ is updated as follows

$$\mathbf{O}^{(t)} = \sigma\left(\mathbf{b}^{\mathrm{O}} + \mathbf{W}_h^{\mathrm{O}} \cdot \mathbf{h}^{(h-1)} + \mathbf{W}_x^{\mathrm{O}} \cdot \mathbf{x}^{(t)}\right), \tag{20}$$

$$\mathbf{h}^{(t)} = \mathbf{O}^{(t)} \odot \tanh\left(\mathbf{C}^{(t)}\right), \tag{21}$$

where $\mathbf{O}^{(t)}$ is known as the output gate which uses the sigmoid activation function for gating and the corresponding bias parameters $\mathbf{b}^{\mathrm{O}} \in \mathbb{R}^k$ and weights $\mathbf{W}_h^{\mathrm{O}} \in \mathbb{R}^{k\times k}$ and $\mathbf{W}_x^{\mathrm{O}} \in \mathbb{R}^{k\times m}$. The new hidden state $\mathbf{h}^{(t)}$ is then used to compute what to forget, input, and output by the cell in the next time step. Thus, while both hidden state and cell state control the memory of the network, the cell state carries information about the entire sequence and the hidden state encodes the information about the most recent time step. Similarly to a deep vanilla RNN architecture, in a deep LSTM ANN the hidden state of each LSTM cell is passed to the next layer, as shown in figure 1(a).

Because each of the four gates is based on a perceptron the total number of trainable parameters of the LSTM variant shown in figure 1(c) is $4k(k+m+1)$ and, since $\mathbf{b}^{\mathrm{F}}, \mathbf{b}^{\mathrm{I}}, \mathbf{b}^{\mathrm{O}}$, $\mathbf{W}_h^{\mathrm{F}}$, $\mathbf{W}_h^{\mathrm{I}}$, $\mathbf{W}_h^{\mathrm{O}}$, $\mathbf{W}_x^{\mathrm{F}}$, $\mathbf{W}_x^{\mathrm{I}}$, and $\mathbf{W}_x^{\mathrm{O}}$ are shared across all LSTM cells, this is also the total number of trainable parameters of the first LSTM layer comprised of any number of LSTM cells. The total number of trainable parameters of a two-layer LSTM architecture is $4[k_2(k_1+k_2+1)+k_1(k_1+m+1)]$. Thus, such LSTM models contain exactly four times more trainable parameters than the vanilla RNN models with the same number of UNITS (length of the hidden state vector).

2.2.5. GRU

LSTM has become a popular off-the-shelf architecture that effectively solves the vanishing gradient problem. However, LSTM is often criticized for its ad hoc nature. Furthermore, the purpose of its many components is not apparent and there is no proof that LSTM is even the optimal structure. Jozefowicz *et al* [91] showed that the forget gate is crucial element of the LSTM architecture, while the output gate is the least important.

GRU [89–91, 151] ANNs are another example of gated RNNs. GRUs were introduced as a simplified version of LSTM that uses one less gate and, thus, has fewer trainable parameters. The main difference between LSTM and GRU cells is that forget and input gates that appear in the LSTM architecture are combined together in one gate in the GRU cell. Additionally, the hidden state and cell state are combined as well. The information between GRU cells is transferred via the hidden state vector. Due to the reduction in trainable parameters and complexity, models based on GRU cells tend to converge faster.

There exist several variants of a GRU cell. Figure 1(d) illustrates the GRU cell used in this work. The update equations are discussed below. An input vector $\mathbf{x}^{(t)}$ is concatenated with the previous cell hidden state $\mathbf{h}^{(t-1)}$ and passed through the update Z and reset R gates both using the sigmoid activation function

$$\mathbf{Z}^{(t)} = \sigma\left(\mathbf{b}^Z + \mathbf{W}_h^Z \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_x^Z \cdot \mathbf{x}^{(t)}\right), \tag{22}$$

$$\mathbf{R}^{(t)} = \sigma\left(\mathbf{b}^R + \mathbf{W}_h^R \cdot \mathbf{h}^{(t-1)} + \mathbf{W}_x^R \cdot \mathbf{x}^{(t)}\right), \tag{23}$$

where $\mathbf{b}^{Z(R)} = \mathbf{b}_{W_h}^{Z(R)} + \mathbf{b}_{W_x}^{Z(R)} \in \mathbb{R}^k$ are the biases and $\mathbf{W}_h^{Z(R)} \in \mathbb{R}^{k\times k}$, $\mathbf{W}_x^{Z(R)} \in \mathbb{R}^{k\times m}$ are the weights of the update (reset) gate. The update gate controls the update of the memory state $\mathbf{h}^{(t)}$. The reset gate controls the influence of the hidden state $\mathbf{h}^{(t-1)}$ on $\mathbf{G}^{(t)}$ introducing additional nonlinear effect in the relationship between past and future states. The two gates can individually 'ignore' parts of the hidden state vector [40].

Next, the input vector $\mathbf{x}^{(t)}$ is concatenated with the element-wise product of the hidden state vector $\mathbf{h}^{(t-1)}$ and the output of the reset gate which, after passing through hyperbolic tangent activation function, produces the so-called proposed new candidate state

$$\mathbf{G}^{(t)} = \tanh\left(\mathbf{b}^G + \mathbf{W}_h^G \cdot \left(\mathbf{R}^{(t)} \odot \mathbf{h}^{(t-1)}\right) + \mathbf{W}_x^G \cdot \mathbf{x}^{(t)}\right), \tag{24}$$

where $\mathbf{b}^G = \mathbf{b}_{W_h}^G + \mathbf{b}_{W_x}^G \in \mathbb{R}^k$, $\mathbf{W}_h^G \in \mathbb{R}^{k\times k}$, and $\mathbf{W}_x^G \in \mathbb{R}^{k\times m}$ are the corresponding bias vector and weights. The hidden state is updated by combining the previous cell hidden state $\mathbf{h}^{(t-1)}$ with the output of the update gate as well as with the product of the output of the update gate and the proposed candidate gate

$$\mathbf{h}^{(t)} = \left(1 - \mathbf{Z}^{(t)}\right) \odot \mathbf{h}^{(t-1)} + \mathbf{Z}^{(t)} \odot \mathbf{G}^{(t)}. \tag{25}$$

The total number of trainable parameters of a single GRU cell described above, as implemented in TENSORFLOW software library, is $3k(k+m+2)$. Note the use of the two separate sets of biases $\mathbf{b}_{W_h}$ and $\mathbf{b}_{W_x}$. The total number of trainable parameters is independent of the number of GRU cells due to parameter sharing. Two-layer GRU models contain $3[k_2(k_2+k_1+2)+k_1(k_1+m+2)]$ trainable parameters. Thus, given the same k_1, k_2, and m, GRU models contain approximately three times more trainable parameters than the corresponding vanilla RNN models and *ca.* 1/4 less trainable parameters than LSTM models with the same number of layers.

2.2.6. BRNNs

In all the RNNs, described above the state at time T uses information from the past $\left(x^{(t)}\right), t = 1, \ldots, T-1$ as well as the current state $x^{(T)}$ to make a prediction. Such RNNs are said to have 'causal' structure [40] and are called unidirectional RNNs. RNN architectures that combine an RNN that moves forward through time

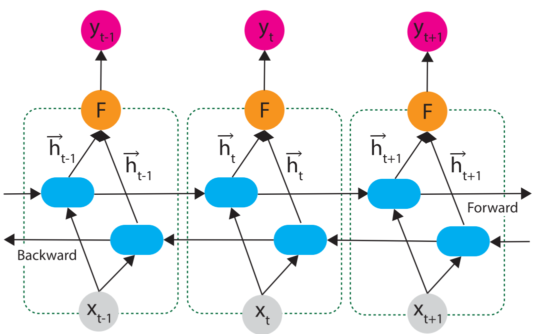

Figure 2. Bidirectional recurrent neural networks. Blue rounded rectangles represent cells and x_t is the tth element of an input sequence **x**. The input sequence **x** is processed by the two disconnected RNNs from beginning to the end and vice versa. Correspondingly, two hidden state vectors: forward $\overrightarrow{\mathbf{h}}^{(t)}$ and backward $\overleftarrow{\mathbf{h}}^{(t)}$ are updated and passed between the cells of the same layer.

from $t = 1$ to $t = T$ with another RNN that moves backward through time from $t = T$ to $t = 1$ are called BRNNs [40, 57] (figure 2). This is realized by duplicating each recurrent layer in the network. The two resulting layers have separate forward $\overrightarrow{\mathbf{h}}$ and backward $\overleftarrow{\mathbf{h}}$ hidden state vectors. Forward and backward layers are not connected to each other. An input time sequence is provided in the chronological order to the first RNN layer and is fed in the reversed chronological order to the second RNN layer. Applying RNNs twice increases the amount of input information available to the network and leads to better capturing long-term dependencies and, thus, improves the accuracy of the model [93].

Bidirectional recurrent ANNs can be built from the vanilla RNN, LSTM, and GRU cells resulting in BRNN, BLSTM [92], BGRU models [68], respectively. In each case the update equations are similar to equations (15)–(25) but applied separately to forward and backward hidden state vectors. For example, for BRNN with tanh activation function, $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$, assuming they have the identical size, $\overrightarrow{\mathbf{h}}, \overleftarrow{\mathbf{h}} \in \mathbb{R}^k$, are updated as follows

$$\overrightarrow{\mathbf{h}}^{(t)} = \tanh\left(\mathbf{b}_{\overrightarrow{h}} + \mathbf{W}_{\overrightarrow{h}} \cdot \overrightarrow{\mathbf{h}}^{(t-1)} + \mathbf{W}_{x\overrightarrow{h}} \cdot \mathbf{x}^{(t)}\right), \tag{26}$$

$$\overleftarrow{\mathbf{h}}^{(t)} = \tanh\left(\mathbf{b}_{\overleftarrow{h}} + \mathbf{W}_{\overleftarrow{h}} \cdot \overleftarrow{\mathbf{h}}^{(t+1)} + \mathbf{W}_{x\overleftarrow{h}} \cdot \mathbf{x}^{(t)}\right), \tag{27}$$

where $\mathbf{b}_{\overrightarrow{h}}, \mathbf{b}_{\overleftarrow{h}} \in \mathbb{R}^k$ are the bias parameters for the forward and backward hidden states, respectively; $\mathbf{W}_{\overrightarrow{h}}, \mathbf{W}_{\overleftarrow{h}} \in \mathbb{R}^{k\times k}$ and $\mathbf{W}_{x\overrightarrow{h}}, \mathbf{W}_{x\overleftarrow{h}} \in \mathbb{R}^{k\times m}$ are the coefficient matrices. In deep BRNN architectures, forward and backward hidden states of a previous layer are combined

$$\mathbf{y}^{(t)} = \mathcal{G}\left(\overrightarrow{\mathbf{h}}^{(t)}, \overleftarrow{\mathbf{h}}^{(t)}\right), \tag{28}$$

and passed to the following layer. In equation (28) $\mathcal{G}$ is a function that combines the two hidden state vectors. It can be a concatenating function (this work), element-wise addition, multiplication, or averaging [145, 152]. Thus every hidden RNN layer receives and input from both forward and backward layers of the preceding layer.

The total number of trainable parameters of the first layer of BRNN is exactly twice the number of trainable parameters in the first layer of a unidirectional RNN with the same length of the hidden state vector and same type of RNN cell. In a deep BRNN architecture an ith hidden layer with m RNN cells and the length of hidden vectors k_i outputs to the next RNN layer a tensor $\mathbf{Z} \in \mathbb{R}^{2k_i\times m}$, assuming $\mathcal{G}$ is a concatenating function. The next $(i+1)$ layer uses $2k_{i+1}(2k_i + k_{i+1} + 1)$ trainable parameters, where k_{i+1} is the length of the hidden state vector of the $(i+1)$th layer. Therefore, deep BRNNs, in general, use more than twice the number of trainable parameters than the corresponding unidirectional RNN with the same number of cells and length of the hidden state vector.

2.2.7. CRNNs

CRNNs are deep ANNs that combine convolutional layers with recurrent layers. In this work, we build two-layer CRNN models by adding one recurrent layer (vanilla RNN, GRU, LSTM) to 1D CNN layer resulting in three models denoted as CRNN, convolutional GRU (CGRU), and convolutional LSTM (CLSTM). The update equations of each of these types of neural networks are exactly those of the corresponding individual layers described above. For example, in the CRNN architecture, the first layer (1D CNN) processes input sequence of length T through convolution, or rather, cross-correlation, operation as given by equations (10) and (11) and generates the output tensor $\mathbf{Z}^{(1)}$ formed by the output of K_1 kernels (filters) each of size $M_1 = \lfloor (T - k_1 + 2p_1)/s_1 \rfloor + 1$, where s_1 is the stride and p_1 is the zero padding. The output tensor $\mathbf{Z}^{(1)}$ is then passed to an RNN layer which is comprised of M_1 vanilla RNN cells with the hidden state vector of (user-specified) length k_2. Each RNN cell receives an input $x^{(t)} \equiv Z^{(1)}_{t,1:K_1}, t \in 1, \ldots, M_1$ from the preceding 1D CNN layer and processes it according to the corresponding update equations.

In this work the recurrent layer of CRNN models is configured to output the whole sequence of hidden states from all RNN cells, $\mathbf{Z}^{(2)} \in \mathbb{R}^{M_1 \times k_2}$ which is then passed to fully-connected layers. The total number of trainable parameters of the two-layer architecture described above is $K_1(k+1) + N_{\mathrm{RNN}}$, where k_1 is the size of the kernel of 1D CNN layer and N_{RNN} the number of trainable parameters of an RNN layer.

2.2.8. CBRNNs

It is possible to fuse a 1D CNN layer and a BRNN layer. Resulting architectures are called CBRNNs. Such models are expected to combine the benefits of CRNNs with the improved description of long-term dependencies pertinent to BRNNs. In this work, three convolutional bidirectional ANN models are build by combining one 1D CNN layer with each of the three recurrent ANN architectures discussed above: the vanilla RNN, GRU, and LSTM. The resulting models are denoted as CBRNN, convolutional bidirectional GRU (CBGRU), and convolutional bidirectional LSTM (CBLSTM). Similarly to CRNNs the update equations of convolutional bidirectional ANNs can be deduced from the corresponding update equations of 1D CNN and BRNNs given above. Specifically for CBRNN, an input sequence is first passed through an 1D CNN layer where it is transformed according to equations (10) and (11) into an output tensor $\mathbf{Z}^{(1)}$ which is formed by the output of K_1 kernels (filters) each of size $M_1 = \lfloor (T - k_1 + 2p_1)/s_1 \rfloor + 1$, where s_1 is the stride and p_1 is the zero padding. Then, $\mathbf{Z}^{(1)}$ is fed separately into forward and backward vanilla RNN layers where the corresponding forward and backward hidden states are updated as shown in equations (26) and (27). In the two-layer architecture described above, the total number of trainable parameters is simply the sum of the total number of trainable parameters of the 1D CNN layer and the total number of trainable parameters of a bidirectional recurrent layer.

2.2.9. KRR

In KRR [153–155] the approximating function $f(\mathbf{x})$ for a vector of input values $\mathbf{x}$ is defined as

$$f(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i k(\mathbf{x}, \mathbf{x}_i), \tag{29}$$

where N is the number of training points and $\boldsymbol{\alpha} = \{\alpha_i\}$ is a vector of regression coefficients. The covariance function $k(\mathbf{x}, \mathbf{x}_i)$, commonly referred to as a kernel function or simply kernel, can be understood as a similarity measure between two vectors $\mathbf{x}$ and $\mathbf{x}_i$ from the input space. The kernel performs an implicit mapping to a higher-dimensional feature space.

Because input time-sequences employed in this work have the same length we focus on standard kernels [116]. One of the most common kernel functions is the Matérn kernel [117, 156], which in the MLatom software package [154, 158] used in this work is defined as: [157]

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2}{\sigma}\right) \sum_{k=0}^{n} \frac{(n+k)!}{(2n)!} \binom{n}{k} \left(\frac{2\|\mathbf{x}_i - \mathbf{x}_j\|_2}{\sigma}\right)^{n-k}, \tag{30}$$

where σ is a positive hyperparameter which defines the characteristic length scale of the covariance function, n is a non-negative integer, and $\|\ldots\|_2$ is the Euclidian distance which is taken to be the L^2 norm. For $n = 0$ the Matérn covariance function reduces to exponential kernel function [117, 156]

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2}{\sigma}\right). \tag{31}$$

Another popular choice of a covariance function is the squared exponential (Gaussian) kernel function [156]

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{2\sigma^2}\right). \tag{32}$$

Because quantum dissipative dynamics often resembles periodically-decaying time-series, we also test a decaying periodic kernel (our adaptation based on [156, 159]):

$$k\left(\mathbf{x}_i,\mathbf{x}_j\right) = \exp\left(-\frac{\|\mathbf{x}_i-\mathbf{x}_j\|_2^2}{2\sigma^2} - \frac{2}{\sigma_p^2}\sin^2\left(\frac{\pi}{p}\|\mathbf{x}_i-\mathbf{x}_j\|_2\right)\right), \tag{33}$$

where p is the period and σ_p is a length scale for the periodic term (both are hyperparameters).

Given the kernel function, the regression coefficients $\boldsymbol{\alpha}$ are found by minimizing a squared error loss function

$$\min_{\alpha}\sum_{i=1}^{N}\left(f(\mathbf{x}_i)-y_i\right)^2+\lambda\boldsymbol{\alpha}^T\mathbf{K}\boldsymbol{\alpha}, \tag{34}$$

where $\mathbf{y}=\{y_i\}$ is the target output vector, $\mathbf{K}\in\mathbb{R}^{N\times N}$ is the kernel matrix with elements $K_{ij}=k\left(\mathbf{x}_i,\mathbf{x}_j\right)$ and λ denotes a non-negative regularization hyperparameter. In equation (34), the second term is usually added to prevent KRR model from assigning large weight to a single point. The optimization of parameters (regression coefficients $\boldsymbol{\alpha}$) amounts to solving a system of linear equations

$$(\mathbf{K}+\lambda\mathbf{I})=\mathbf{y}, \tag{35}$$

for which analytical solution is known. Here $\mathbf{I}$ is the identity matrix. The computational scaling of solving this system is $\mathcal{O}(N^3)$ [156, 157].

KRR is a kernelized version of ridge regression and when linear kernel function

$$k\left(\mathbf{x}_i,\mathbf{x}_j\right)=\mathbf{x}_i^T\mathbf{x}_j \tag{36}$$

is used, KRR becomes equivalent to ridge regression, i.e. the approximating function $f(\mathbf{x})$ is simply a multiple linear regression with regression coefficients $\boldsymbol{\beta}$ shrunk (see the second term in equation (34)) using regularization:

$$f(\mathbf{x})=\sum_{i=1}^{N}\alpha_i\mathbf{x}_i^T\mathbf{x}=\sum_{i=1}^{N}\alpha_i\sum_{s=1}^{T}x_{is}x_s=\sum_{s=1}^{T}\left(x_s\sum_{i=1}^{N}\alpha_i x_{is}\right)=\sum_{s=1}^{T}\beta_s x_s. \tag{37}$$

As shown in the above equation, regression coefficients $\boldsymbol{\beta}$ can be conveniently derived from $\boldsymbol{\alpha}$ coefficients and training input data and are printed out by MLatom.

3. Computational details

3.1. Data sets for training, validation, and testing

The data set used in this work was generated as detailed below [33]. Firstly, HEOM calculations for all combinations of the following parameters: $\epsilon/\Delta=\{0,1\}$, $\lambda/\Delta=\{0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0\}$, $\omega_c/\Delta=\{1,2,3,4,5,6,7,8,9,10\}$, and $\beta\Delta=\{0.1,0.25,0.5,0.75,1\}$, were performed with QuTiP software package [160]. In our calculations we set $\Delta=1.0$. The total propagation time was $t_{\max}\Delta=20$ and the HEOM integration time-step was set to $t\Delta=0.05$. In total, 1000 HEOM calculations, 500 for symmetric ($\epsilon/\Delta=0$) and 500 for asymmetric ($\epsilon/\Delta=1$) spin-boson Hamiltonian were performed. Time-evolved reduced density matrices (RDM) are saved every $\mathrm{d}t\Delta=0.1$. Secondly, $\langle\hat{\sigma}_z(t)\rangle$ are calculated from RDMs and processed into shorter sequences of length T by window slicing [32, 33, 125]. Namely, for a time series $\mathbf{x}=\left(x^{(1)},\dots,x^{(L)}\right)$, where $\langle\hat{\sigma}_z(t)\rangle$ is denoted by $x^{(t)}$ for compactness, a slice is a subset of the original time series defined as $\mathbf{s}_{i:j}=\left(x^{(i)},\dots,x^{(j)}\right)$, $1\leqslant i\leqslant j\leqslant P$. For a given time series $\mathbf{x}$ of length L, and the length of the slice P, a set of $L-P+1$ sliced time series $\{\mathbf{s}_{1:P},\mathbf{s}_{2:P+1},\dots,\mathbf{s}_{L-P+1:L}\}$ is generated. Finally, the total data set $\mathcal{D}=\{(\mathbf{x}_i,y_i)\}_{i=1}^{N}$ containing time series $\mathbf{x}_i$ and their corresponding labels y_i is obtained by setting $1,\dots,T$ elements of each slice, with $T=P-1$, to an input time-series $\mathbf{x}_i$ and the last (Pth) element of each slice to the associated label y_i.

In general, the size of the window $P-1$, or equivalently T, should be treated as a hyperparameter but, following previous work [33], we set $T=0.2L$. The window slicing is applied to all 1000 RDMs obtained in HEOM calculations with different system and system-bath parameters. For each set of parameters the initially calculated set of time-evolved $\langle\hat{\sigma}_z(t)\rangle$ with $L=t_{\max}/\mathrm{d}t=200$ generates 160 data points for the data set with $T=41$ (including $t\Delta=0$ point).

From the raw HEOM data set of 1000 trajectories, 100 randomly chosen trajectories are taken as the hold-out test set, which is used for testing and generating the results presented in section 4. The remaining set of 900 trajectories are transformed into 144 000 trajectories by window slicing described above. In total 72 000 short-time $\langle\hat{\sigma}_z(t)\rangle$ trajectories for symmetric and 72 000 for asymmetric spin-boson models were generated. Each trajectory has a time length of $t\Delta + \mathrm{d}t\Delta = 4.1$ ($t\Delta = 4.0$ is a part of a trajectory used as input vector $\mathbf{x}_i$ and the last point at time $t\Delta + \mathrm{d}t\Delta = 4.1$ is used as a label y_i). Following previous similar works [32, 33] the input data was not normalized. This data set of supervised trajectories is the training set which is randomly partitioned into two subsets: a sub-training set, which contains 80% of the data and a validation set containing 20% of the data. ML model parameters (weights and biases in ANNs and regression coefficients in KRR) are (initially) fitted on the sub-training set and the validation set is used for monitoring the performance of the models (mainly, to prevent overfitting and tune hyperparameters). Thus initially fitted ANN parameters were not further re-fitted and resulted in final ANN models which were tested on a hold-out test set. In case of KRR models, we did not stop at this initial training and after hyperparameter optimization, we used these hyperparameters to fit KRR regression coefficients on the entire training set including the validation set, resulting in our final KRR models used for performance evaluation. The performance of KRR models with the initial regression coefficients fitted only on the sub-training set is similar to the final KRR models trained on the entire training set (appendix A). By 'fitting' in case of ANN we mean backpropagation optimization of parameters, while in case of KRR – analytical solution of the regression problem formulated for a given set (training or sub-training). These are typical ways of training ANN and KRR models, respectively, which are different due to differences in the formalism and training procedures of these types of models. To emphasize, in all cases, ANN models, initial and final KRR models, the validation set is thus also used for training, either directly (final KRR models) or indirectly (ANN and initial KRR models).

3.2. ANN models

3.2.1. Details of the models

In this work fourteen deep ANN models are built and tested. In general, each ANN model is comprised by the input layer, the total of two convolutional and/or recurrent layers followed by one fully-connected layer, and the output layer. The exception is the FFNN model which is comprised of an input layer, and two fully-connected hidden layers followed by the output layer. The fully-connected layer in each model, except for FFNN, has 256 neurons and the rectified linear unit (ReLU) function $f(z) = \max(0, z)$ is used as the activation function. Fixing the properties of fully-connected layer allows to compare the performance of recurrent and convolutional layers. The output layer contains one neuron with the linear activation function $f(z) = z$. The details of all ANN models studied in the present Article are summarized below.

- 1D CNN model contains two 1D CNN layers followed by a MaxPooling, one fully-connected and output layers. ReLU is used as the activation function in each 1D CNN and fully-connected layers. For the MaxPooling layer a pool size $k_p = 2$ (see equation (13)) is used. The stride of $s = 1$ and zero padding are used in both 1D CNN and MaxPooling layers. The number of filters and the filter sizes of each layer are optimized using Particle Swarm Optimization (PSO) algorithm as described in section 3.2.2.
- FFNN model contains two hidden fully-connected layers followed by the output layer. The ReLU activation function is used.
- Three recurrent ANN models comprised of the two recurrent layers of the same type: the vanilla RNN, LSTM, and GRU. The whole sequence of hidden state vectors from all cells is passed from the first to the second recurrent layer as well as from the second recurrent layer to the fully-connected layer.
- Three BRNN models comprised of the two bidirectional recurrent layers of the same type: the vanilla BRNN (denoted simply as BRNN hereafter), BLSTM, and BGRU. The whole sequence of hidden state vectors from all cells is passed from the first bidirectional recurrent layer to the second bidirectional recurrent layer as well as from the second bidirectional recurrent layer to the fully-connected layer.
- Three CRNN models comprised of one 1D CNN layer followed by one recurrent layer of each type: the vanilla RNN, LSTM, and GRU. The resulting models are denoted as CRNN, CLSTM, and CGRU. The recurrent layer returns the whole sequence of hidden state vectors from all cells to the fully-connected layer. The stride $s = 1$, zero padding, and the ReLU activation function are used in the 1D CNN layer. The number of filters and the filter size is the same as in the first layer of the 1D CNN model.
- Three CBRNN models comprised of one 1D CNN layer and one bidirectional recurrent layer of each type: the vanilla RNN, LSTM, and GRU. The resulting models are denoted as CBRNN, CBLSTM, and CBGRU. The stride $s = 1$, zero padding, and the ReLU activation function are used in the 1D CNN layer. The number of filters and the filter size are the same as in the first layer of the 1D CNN model. The bidirectional recurrent layer returns the whole sequence of the forward-backward hidden state vectors from all cells to the fully-connected layer.

3.2.2. Hyperparameter optimization

The goal of this work is to compare the performance of ML models including ANN models with different architectures as described above. Depending on the overall objective, one can envision several approaches for comparing performances of different ANNs. For example, the hyperparameters of each ANN model can be adjusted using grid search, random search, or other optimization techniques, including evolutionary algorithms, to achieve the best performance of each ANN model on a given data set. However, given the fundamentally different types of ANN models considered in the present Article, this approach may result in several models with approximately the same prediction accuracy but requiring drastically different resources such as CPU (or graphics processing unit (GPU)) time and memory. In such case, it might be reasonable to use ANN models that are not the most accurate, but those that provide an acceptable accuracy, require less computational resources and yield faster training and prediction times. The notion of the optimal running time, however, strongly depends on the problem under study. One can also set the desired accuracy and systematically adjust the hyperparameters of each model to achieve the desired accuracy and then analyze the resulting ANN models. This approach, however, requires specifying the fixed accuracy level which might not be achievable for some ML models and relies on *a priori* knowledge of the dynamics of the system of interest.

The following approach is adopted in the present study. We compare the performance of ANN models with approximately the same number of trainable parameters. Hyperparameters of each ANN model described above are scanned using a grid search approach and models with the pre-determined number of trainable parameters are selected and their performance is reported in section 4. The number of trainable parameters is chosen as follows. In [32, 34] we illustrated that deep 1D CNN models can approximate long-time dynamics of a molecular dimer system (spin-boson-like model) and the Fenna–Matthews–Olson photosynthetic complex accurately. Therefore, the 1D CNN model from [32] is taken as a base model. The hyperparameters of this model associated with convolutional layers, specifically, the number and the size of the kernels of each layer, are optimized using PSO algorithm. All other hyperparameters such as the number of neurons of a fully-connected layer and the activation functions of all neurons are fixed to 256 and ReLU correspondingly. PSO calculation is performed with three particles and for 50 steps. At each step of the PSO calculation, three 1D CNN models, one for each particle, are trained and validated using the above mentioned data set. During training, the deviation between predicted $\hat{y}_i$ and reference values y_i is minimized. In this work, we use the mean squared error (MSE) as the loss function in the minimization

$$\mathrm{MSE} = \frac{1}{N_\mathrm{b}} \sum_{i=1}^{N_\mathrm{b}} (y_i - \hat{y}_i)^2 , \tag{38}$$

where N_b is the batch size. Adaptive moment estimation (Adam) algorithm [161] is used with the initial learning rate set to 1.0×10^{-4}. The initial values of the weights are randomly sampled using Xavier initialization [162]. The biases are initialized to zero. KERAS [145] software package with the TENSORFLOW [152] backend was employed for calculations. The batch size is set to $N_b = 64$ and the training for each model is performed for 30 epochs. As will be shown later this is sufficient for the MSE to drop below 10^{-5}–10^{-6}.

For each trained 1D CNN model, the mean absolute error (MAE) for the validation set is calculated and used as the fitness value in the PSO. Then the hyperparameters of 1D CNN models are adjusted based on the algorithmic details of PSO which can be found in appendix B. After 50 steps of PSO, the 1D CNN models stop improving and the PSO calculation is terminated. The optimized number of kernels of the final 1D CNN model is 235 and 125 with the kernel sizes 16 and 7 of the first and second layer, respectively. The 1D CNN model described above contains a total of 530 258 trainable parameters. This number is taken as reference and the hyperaprameters of all other 13 ANN models are selected to generate models with approximately the same number of trainable parameters. Note, however, that according to the formulas given in section 1, it is not possible to set hyperparameters in all models to achieve exactly the desired number of trainable parameters. Therefore, some variation in the number of trainable parameters across the reported ANN models is to be expected.

Even though the number of trainable parameters is straightforwardly connected to the hyperparameters of each ANN architecture, a grid search is performed over hyperparameters with the goal to understand the sensitivity of the model performance to small variations in the hyperparameters. In all models containing (bidirectional) recurrent layers the number of units is scanned from 5 to 100 with the step of 5. Additionally, in all ANN models containing 1D CNN layers and (bidirectional) recurrent layers, the number of kernels and kernel sizes of the 1D CNN layer is taken to be same as in 1D CNN model while the number of units in the recurrent layer is scanned from 5 to 100 in steps of 5. All the models are trained, validated, and tested using the data set for the symmetric spin-boson model. KERAS software package and Adam algorithm is used for all ANN calculations. The initial learning rate is fixed to 1.0×10^{-4}, the batch size is set to 128, and the training for each model is performed for 30 epochs.

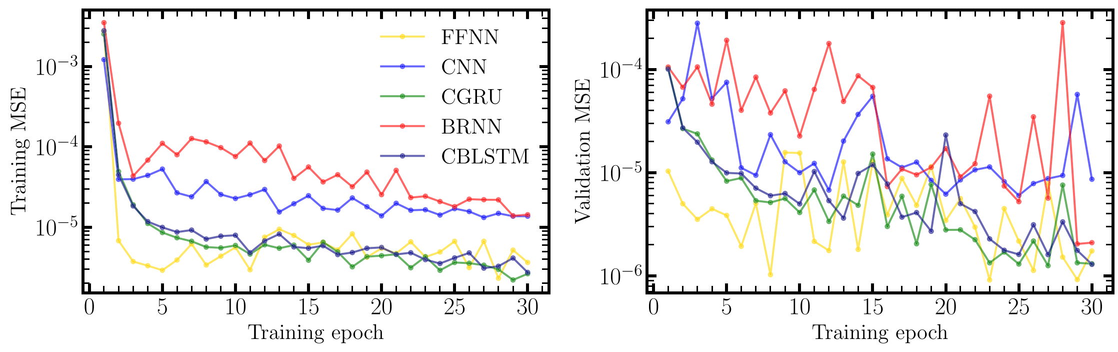

Figure 3. Learning curves for selected ANN models: training (left) and validation (right). Note the logarithmic scale of the vertical axis.

Usually more than one set of hyperparameters generates ANN models with the number of trainable parameters close to the target number. In such cases, models with the number of trainable parameters within $\pm 5\%$ of the reference number are examined. Among selected models, the models that perform significantly different than the average model are discarded. The number of outliers in each case is found to be small. We attribute outliers to finite training time and nonuniform decay of the MSE during the training, which in case of RNNs is a manifestation of the vanishing gradient problem. We will return to this issue in section 4. The model with the closest number of trainable parameters to the target number is saved and its performance is reported.

3.2.3. Training and validation curves

To illustrate the learning process of ANN models, in figure 3 we plot the training and validation curves for several such models. Such curves are a widely used tool to examine the performance of supervised learning algorithms. In general the learning is found to be stable in each case. The apparent noisiness in the data should be attributed to the logarithmic scale of the vertical axis. Several conclusions can be drawn by examining figure 3. Firstly, none of the models overfits which is illustrated in lower and in, general, decaying validation MSE (figure 3 right panel) compared to the training MSEs (figure 3 left panel). Secondly, the learning process is fast even with the learning rate of 1×10^{-4}. It should be noted that decreasing learning rate to 1×10^{-5} does not result in noticeable improvement in accuracy for all the ANN models studied in this work but the increase of the learning rate to 1×10^{-3} usually results in increasing MSE. Furthermore, FFNN model reaches the lowest MSE rapidly in less than 10 training epochs while the MSE of other models e.g. 1D CNN, B(LSTM,GRU) continues to decay over all shown 30 training epochs. The same behavior is manifested in the validation MSEs. This suggests that 30 training epochs used in this work is justified for 1D CNN and other convolutional and recurrent models but seems unnecessarily too many for the FFNN model. This observation further illustrates the efficiency of simple FFNN models. We note that in the case of the BRNN model the validation MSE exhibits large-magnitude changes over the course of training. This can be attributed to the vanishing gradient problem of the vanilla RNNs.

3.3. KRR models

In addition to 14 ANN models, 8 KRR models are investigated in the present Article. Each KRR model differs by the choice of the kernel. The following kernels are studied: the linear kernel denoted as KRR-L (equation (36)), Gaussian kernel (equation (32), KRR-G), exponential kernel equation (31) (KRR-E), Matérn kernel, equation (30) with $n = 1, 2, 3, 4$ (KRR-M1, KRR-M2, KRR-M3, KRR-M4), and a periodic-decaying kernel, equation (33) (KRR-DP). It should be noted that unlike RNNs, KRR (and FFNN) models studied in this work require a fixed-size input. Therefore, to enable the comparison between ANN and KRR models studied in this work, we focused on the input $\langle\hat{\sigma}_z(t)\rangle$ trajectories of the same length. We note that extending RNN models built in the present work to variable size input is straightforward and will be discussed elsewhere.

In KRR models, optimization amounts to finding regression coefficients $\boldsymbol{\alpha}$ as shown in equation (35). Therefore, one can interpret each regression coefficient α_i as a trainable parameter. The total number of such coefficients is the same as the number of elements of the training set, which in this work is 72 000 for both symmetric and asymmetric spin-boson models. This number of trainable parameters of KRR model is far fewer than the target number of trainable parameters for ANN models. We emphasize that the number of trainable parameters for KRR models is, by construction, set to the size of the training set, while it is variable in ANN models. This further makes the faithful comparison of fundamentally different ML approaches studied in this work non-trivial. As will be shown in the following, the fewer number of trainable parameters of KRR does not adversely impact the performance of KRR models.

KRR approaches have analytical solution to optimal parameters (regression coefficients $\boldsymbol{\alpha}$), but they still require adjusting a (small) number of hyperparameters. All KRR models used here contain the regularization hyperparameter λ which is the only hyperparameter in KRR-L. Other KRR models have additional hyperparameters in their kernel functions, i.e. all nonlinear models have length-scale hyperparameter σ, KRR-Mn also have the integer hyperparameter n (not optimized here), KRR-DP has two more hyperparameters compared to KRR-G: p and σ_p. Due to the small number of hyperparameters, they are optimized on a logarithmic grid as described elsewhere [154] for all models except for KRR-DP which has four hyperparameters. Hyperparameters in KRR-DP are optimized using the tree-structured Parzen estimator [163] via interface to the HYPEROPT package [164]. All hyperparameters of KRR models are optimized using grid search method for KRR.

The hyperparameters of each ML model are optimized only for the data set for symmetric spin-boson model as already indicated above. The optimized models are then re-trained on the asymmetric spin-boson data set keeping the hyperparameters unchanged. MLATOM software package [154, 157] is used for all KRR calculations.

4. Results and discussion

4.1. Symmetric spin-boson model

The details and performance of all ANN and KRR models are summarized in tables 1 and 2, respectively. As seen from table 1, the two best performing ANN models on the symmetric spin-boson data set, are the CLSTM and CBLSTM while the two least accurate ANN models are unidirectional RNN and bidirectional vanilla RNN models. Worse performance of simple vanilla RNN layers compared to LSTM and GRU layers is the expected result. It has been observed in other applications that in contrast to LSTM and GRU models, the performance of vanilla RNN models degrades with the increasing length of input sequence [165, 166]. We note that the weak performance of RNN models is despite the increased number of units or length of the hidden state vector compared to other recurrent ANN models.

A better performance of CRNN models is an illustration of the power of sub-sampling of the initial input data which is then more efficiently learned by the LSTM layer. Comparing the unidirectional or bidirectional recurrent ANN models whether taken separately or as the second layer on top of 1D CNN layer confirms the trend. The best performing model is an (C,B,CB)LSTM model, followed by the (C,B,CB)GRU, and (C,B,CB)RNN. It is also worth mentioning that recurrent ANN models based on bidirectional layers do not outperform their unidirectional counterparts. This is observed for both unidirectional recurrent models compared to bidirectional recurrent models as well as when convolutional unidirectional recurrent models are compared to the corresponding convolutional bidirectional recurrent models.

Focusing on KRR models, whose details and performance is illustrated in table 2, we note that these models generally outperform ANN models with the exception of a KRR model with linear kernel. Furthermore, the KRR-L is the worst performing of all ML models studied in the present work. This is, perhaps, not surprising because it is expected that quantum dynamics of such a complex system as the spin-boson model is highly non-trivial and cannot be captured by simple linear regression.

The differences in accuracy between ML models noted above are, however, not significant in the case of symmetric spin-boson model. This is illustrated in figure 4 where the exact population difference $\langle\hat{\sigma}_z(t)\rangle$ is compared to the ML-predicted $\langle\hat{\sigma}_z(t)\rangle$. We stress that only short $\langle\hat{\sigma}_z(t)\rangle$ trajectory of $t\Delta = 4$ is used as an input. The rest of the dynamics is predicted recursively as done in [32, 33]. Figure 4 compares the performance of the 1D CNN model, whose number of trainable parameters is used as a reference for other ANN models, to the best and the worst performing ANN and KRR models. One notices that the prediction accuracy of the KRR-L model degrades slowly over time but still remains acceptable. This can only be seen in figures 4(a) and (b) where the chosen system and bath parameters generate the oscillatory dynamics of the RDM. In the cases of incoherent relaxation dynamics KRR-L results are indistinguishable from the exact HEOM dynamics.

We stress that even the worst performing ML model, KRR-L, as illustrated in Figure 4 still provides an acceptable accuracy. Therefore, we conclude that all ML models benchmarked in the present article can be trained to provide a sufficient long-time prediction accuracy. One suspected drawback of the recursive propagation approach proposed in [32, 33] is that the prediction error would grow over many time steps possibly leading to significant accuracy loss in the long-time dynamics. However, as shown above such errors can be made small enough to make long-time predictions reliable all the way until the dynamics reaches equilibrium.

Table 1. Hyperparameters, mean absolute prediction errors, total number of trainable parameters, training, and average single step prediction times for all ANN models studied in this work. For each recurrent layer the number of units is shown. The number of kernels X and kernel sizes Y for each convolutional layer are shown in parenthesis as (X, Y).

Model	Trainable parameters	Layers		Mean absolute error		Time (s)	
		Layer 1	Layer 2	Symmetric	Asymmetric	Training	Prediction
1D CNN	530 258	(235,16)	(125,7)	1.55×10^{-3}	4.84×10^{-2}	465	3.8
FFNN	520 045	754	646	1.32×10^{-3}	3.70×10^{-2}	82	3.3
Recurrent Neural Networks							
LSTM	528 577	15	49	1.58×10^{-3}	2.35×10^{-2}	623	6.1
GRU	553 453	60	50	2.05×10^{-3}	2.57×10^{-2}	668	4.4
RNN	535 468	65	50	3.00×10^{-3}	6.17×10^{-2}	302	4.2
Convolutional Recurrent Neural Networks							
CLSTM	501 965	(28,16)	71	1.17×10^{-3}	2.50×10^{-2}	279	4.1
CGRU	515 806	(55,16)	73	1.38×10^{-3}	2.14×10^{-2}	294	4.7
CRNN	513 673	(243,16)	73	1.46×10^{-3}	3.61×10^{-2}	197	5.4
Convolutional Bidirectional Recurrent Neural Networks							
CBLSTM	568 022	(109,16)	39	1.17×10^{-3}	2.84×10^{-2}	333	3.8
CBGRU	514 860	(55,16)	37	1.54×10^{-3}	3.68×10^{-2}	325	5.3
CBRNN	508 842	(297,16)	36	2.50×10^{-3}	3.58×10^{-2}	256	3.9
Bidirectional Recurrent Neural Networks							
BLSTM	511 809	6	24	2.12×10^{-3}	2.56×10^{-2}	635	5.2
BGRU	534 991	14	25	2.28×10^{-3}	2.48×10^{-2}	1109	6.7
BRNN	511 959	37	24	6.95×10^{-3}	4.27×10^{-1}	396	5.4

Table 2. Mean absolute prediction errors, training, and average single step prediction times of all KRR models used in this work. KRR-L, KRR-G, KRR-DP, KRR-E, and KRR-Mn ($n = 1, 2, 3, 4$) denote kernel ridge regression models with linear kernel, Gaussian kernel, decaying-periodic kernel, exponential kernel, and Matérn kernel with $n = 1, 2, 3, 4$, respectively.

Model	Trainable parameters	Mean absolute error		Time (s)	
		Symmetric	Asymmetric	Training	Prediction
KRR-L	72 000	1.2×10^{-2}	6.5×10^{-2}	196	1.6
KRR-G	72 000	4.7×10^{-4}	1.2×10^{-3}	220	1.6
KRR-DP	72 000	4.3×10^{-4}	2.0×10^{-3}	257	1.4
KRR-E	72 000	2.1×10^{-3}	3.3×10^{-3}	222	1.6
KRR-M1	72 000	2.4×10^{-4}	1.3×10^{-3}	260	1.6
KRR-M2	72 000	2.2×10^{-4}	2.7×10^{-3}	259	1.7
KRR-M3	72 000	2.0×10^{-4}	2.3×10^{-3}	279	1.7
KRR-M4	72 000	2.3×10^{-4}	2.1×10^{-3}	273	1.7

4.2. Asymmetric spin-boson model

The results reported so far are encouraging but not discriminative. To unravel the differences between the studied ML models we devise a more stringent test. All the ANN and KRR models are re-trained on the asymmetric spin-boson model data set without any hyperparameter adjustment. The increased difficulty of this test stems from the richer dynamics of the asymmetric spin-boson model which might require more training parameters or even different ANN architecture (more hidden layers, longer hidden state vectors, and memory T). Additionally, from a practical standpoint it is highly desirable to avoid optimization of yet another ML model for asymmetric spin-boson system unless necessary, particularly, if one needs to perform large-scale calculations for many different values of the energy bias ϵ. The purpose of the following discussion is to assess the viability of this approach.

The proposed test clearly indicates that some models predict the long-time dynamics much more accurately than others. The performance of each model on the hold-out test set is reported in tables 1 and 2. There is a noticeable increase in the mean absolute error (MAE) of all studied ML models compared to the symmetric spin-boson model. Generally, KRR models are more robust than ANN models. On average, the MAE of KRR models increases by a factor of 6. KRR models based on Mátern kernels with $n = 2, 3, 4$ exhibit

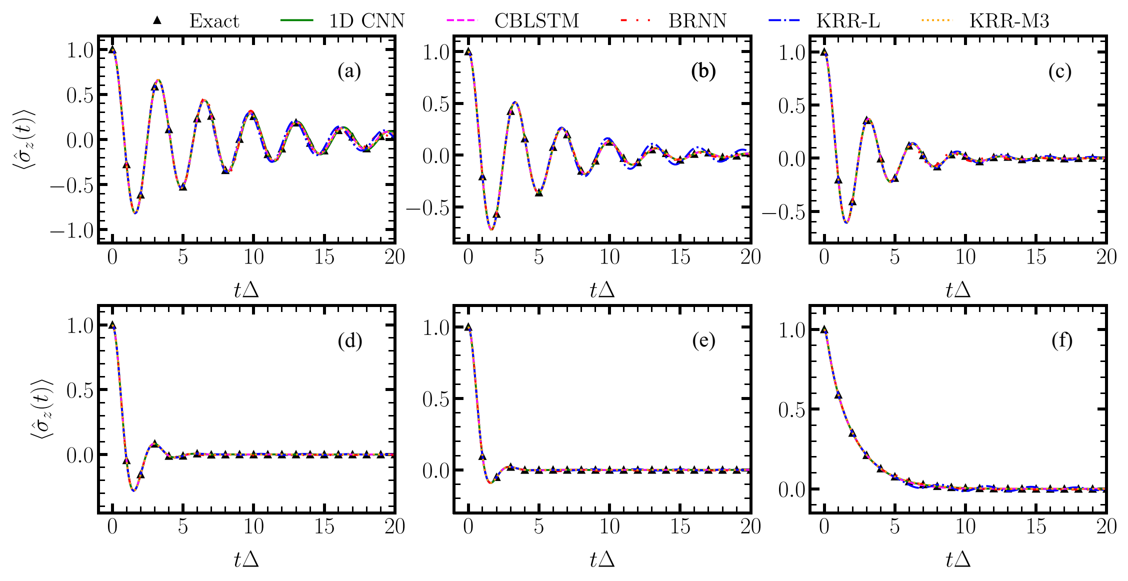

Figure 4. Expectation values $\langle\hat{\sigma}_z(t)\rangle$ for symmetric spin-boson model as a function of time. Results predicted by an indicated ML model are compared to the HEOM results (black triangles) for the following parameters: (a) $\lambda = 0.2, \omega_c = 8.0, \beta = 1.0$; (b) $\lambda = 0.4, \omega_c = 10.0, \beta = 1.0$; (c) $\lambda = 0.2, \omega_c = 10.0, \beta = 0.25$; (d) $\lambda = 0.1, \omega_c = 4.0, \beta = 0.1$; (e) $\lambda = 0.8, \omega_c = 3.0, \beta = 1.0$; (f) $\lambda = 1.0, \omega_c = 2.0, \beta = 0.1$. All parameters are in the units of Δ.

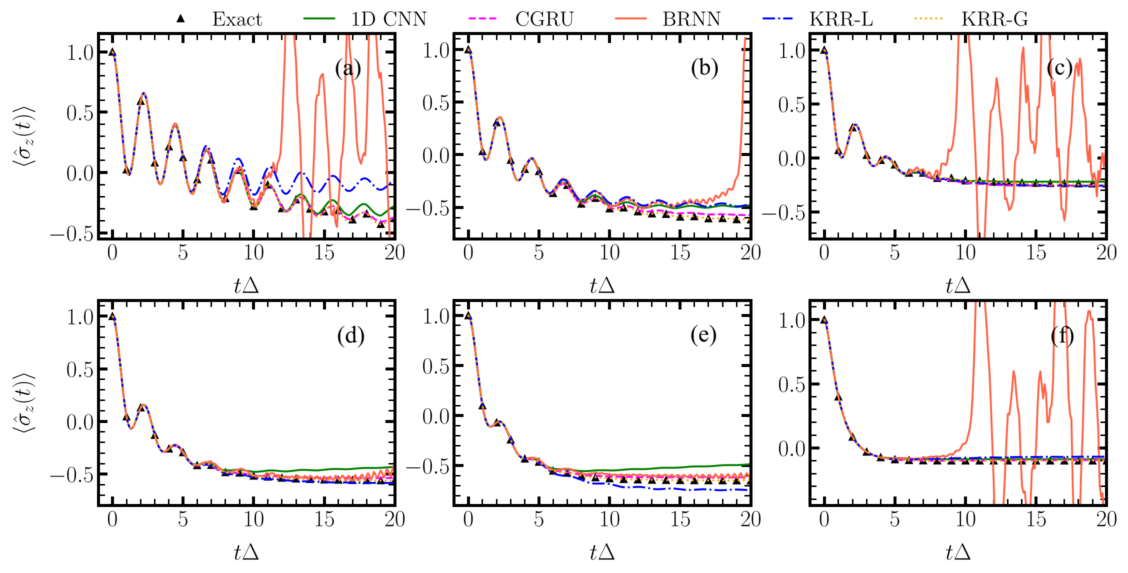

Figure 5. Expectation values $\langle\hat{\sigma}_z(t)\rangle$ for asymmetric spin-boson model with $\epsilon = 1$ as a function of time. Results predicted by an indicated ML model are compared to the HEOM results (black triangles) for the following parameters: (a) $\lambda = 0.1, \omega_c = 6.0, \beta = 0.75$; (b) $\lambda = 0.3, \omega_c = 8.0, \beta = 1.0$; (c) $\lambda = 0.2, \omega_c = 10.0, \beta = 0.25$; (d) $\lambda = 0.4, \omega_c = 8.0, \beta = 0.75$; (e) $\lambda = 0.8, \omega_c = 10.0, \beta = 1.0$; (f) $\lambda = 0.7, \omega_c = 10.0, \beta = 0.1$. All parameters are in the units of Δ.

the most significant, ∼10-fold, performance drop. The average decrease in performance of ANN models is ∼18-fold. Interestingly, the most significant performance reduction, among the ANN models, is observed for the models that do not contain recurrent layers: 1D CNN and FFNN while models containing bidirectional recurrent layers revealed to be the most robust ANN models. On average the MAE of B(LSTM, GRU, RNN) models decreased by a factor of ∼10 while, for example, the FFNN model became ∼30 times less accurate for the asymmetric spin-boson model.

Figure 5 shows ML-predicted dynamics of $\langle\hat{\sigma}_z(t)\rangle$ for the most and least accurate ANN and KRR models for the asymmetric spin-boson model as well as for the 'reference' ANN model, 1D CNN. Analogously to the symmetric spin-boson model, BRNN model is found to be the least accurate. The difference is, however, more dramatic. Clearly, BRNN model is overall no longer acceptable ANN model even though it provides an accurate prediction for some parameters, see figures 5(d) and (e), but even in these cases one can easily distinguish an unphysical oscillatory behavoir building up beyond relatively short times of $t\Delta \approx 10$. All other ANN models provide more accurate and stable predictions without noise and unphysical artifacts. The CGRU model is the most accurate matching the HEOM dynamics nearly exactly. The worst KRR model is, once again, the one with the linear kernel. It is however, able to capture the dynamics qualitatively, and in some, cases fairly accurately. It seems to be less reliable in predicting the oscillatory dynamics typically observed in low temperature and small reorganization energy regimes. The most accurate KRR model is the one with the Gaussian kernel. This model predicts the reference HEOM dynamics very accurately. The promise of the KRR-G model has been already pointed out in [33].

4.3. Training and prediction times

The accuracy and robustness of ML models are clearly very important but our analysis would be incomplete without discussing the timings associated with all studied ML models. In large-scale applications, such as computational materials screening that requires optimization with respect to long-time value of a dynamically-changing physical property (e.g. population transfer, charge-transfer efficiency, etc) a separate quantum-dynamical calculation for each candidate is required in the search for the optimal solution [34, 167]. These studies require huge, often *a priori* unknown, number of repetitive time-consuming quantum-dynamical calculations. If a fraction of each calculation, beyond short initial time, can be replaced with ML this would significantly speed-up the screening.

All models are trained on the same machine with 2 × 20 C Intel Xeon Gold 6230 2.1 GHz processor and 192 Gb DDR4 memory. Training times are reported in tables 1 and 2 (only times for training are shown, in practice, the cost is higher if the hyperparameter optimization is performed). We note that training of ANNs can be greatly accelerated by using GPUs. While we have not used GPUs in the present study, we anticipate our conclusions, but not the actual values of training and prediction times, to hold the same as if our ML models were trained on GPUs. One should also be mindful that different software packages are used for ANN and KRR models which makes the direct comparison somewhat dubious. Nonetheless both KERAS and MLATOM are state-of-the-art software widely used in the community so we will proceed with the comparison.

According to table 1, the accuracy of FFNN and CBGRU models for the asymmetric spin-boson model is the same but the training time of the FFNN model is nearly 4 times shorter. Furthermore, for the symmetric spin-boson Hamiltonian, FFNN model outperforms all ANN models comprised of only recurrent layers but shows similar performance to CRNNs and BRNNs and it does so faster: FFNN model is 3 times faster to train than the best performing CLSTM and CGRU models.

Continuing the comparison of the training times we note that, due to their simplicity, the vanilla (unidirectional) RNN models are also fast to train. Oppositely, BRNNs of all kinds are among the slowest models to train. Given that using bidirectional architecture does not reduce the error compared to their unidirectional counterparts while the increased training time of bidirectional layers makes such models disfavored for, at least, the problem of predicting the long-time dynamics of the spin-boson problem. This result is interesting given that BRNNs have some success in other domains as described in Introduction.

Training times of KRR models are relatively fast compared to most of the ANN models except for FFNN and CRNN. The average single-step prediction times for each ANN and KRR models are calculated on the same machine used for training and are shown in the last column of tables 1 and 2. It is clear that KRR models take less time to make a single time-step prediction than ANN models, although this can be attributed to the software nuances. Expectedly, the FFNN model is the fastest ANN model while BGRU is the slowest. Similar to the training time we observe that (C,B)GRU models are slower than (C,B)LSTM models which is counterintuitive given that GRU cell is less complex than the LSTM one. We attribute this inconsistency to the implementation details of both cells in KERAS. Overall, recurrent layers require more time to evaluate than other layers as should be anticipated. BRNNs take even more time to make a prediction compared to their unidirectional counterparts. CRNN models make predictions faster compared to their two-layer recurrent counterparts as they replace one computationally more expensive recurrent layer with a cheaper 1D CNN layer.

4.4. Model efficiency

To illustrate the tradeoff between accuracy and training/prediction time, in figure 6 the training and single-step prediction times are plotted against the MAE for each ML model built in the present work. For ease of illustration, the models of similar kind are highlighted with the same color, e.g. all KRR models are depicted by green circles, magenta color is used for all convolutional recurrent models, etc. Clearly, KRR models are the most efficient models showing high accuracy and requiring the shortest amount of time to train and make a prediction. Among the ANN models, convolutional recurrent models are the most efficient. According to figure 6 convolutional bidirectional NNs, specifically, CBLSTM constitute the second most

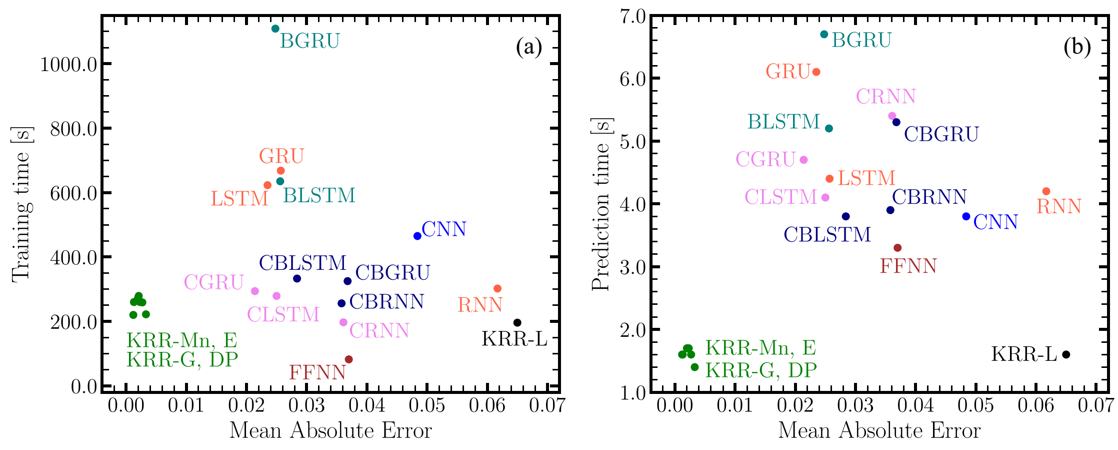

Figure 6. The mean absolute single-step prediction errors of $\langle\hat{\sigma}_z(t)\rangle$ of machine learning models studied in this work plotted against the corresponding training (a) and single-step prediction (b) times. Bidirectional recurrent neural network model is not shown because its error is much too big compared to other models (see table 1 and figure 5).

efficient set of ANN models. In contrast, because of the longer training times and not much improved accuracy, bidirectional recurrent ANN models are least efficient.

5. Conclusions and outlook

We performed a benchmark study of 22 supervised ML methods comparing their ability to accurately forecast long-time dynamics of a two-level quantum system linearly coupled to harmonic bath. We illustrate that many of the studied ML methods can achieve a good agreement with the exact population dynamics. Our study reveals that if the memory time of a quantum dynamical system under study is known, the models based on KRR are the most accurate and should be preferred. The commonly employed Gaussian kernel is confirmed to be the most efficient yielding the accuracy on a par with other nonlinear kernels while requiring somewhat less than training time.

An excellent performance of KRR models tested here compared to ANN models specifically designed for a task of learning time-series may come as a surprise. While it may be due to the nature of an investigated problem (deterministic time series), kernel-based methods such as KRR and closely related Gaussian process regression are known to outperform ANNs in different tasks from several independent studies [168–171]. In addition, in some cases kernel-based methods can be considered as infinitely large FFNNs which is a manifestation of the power of kernel-based approaches [156].

CRNNs appear to be the most promising ANN models. One scenario where such models are suitable are the problems when the memory time of the problem is not available and flexible ML models allowing variable size input are needed. Such investigation is beyond the scope of this work and will be performed and reported in future studies. Based on the present study we conclude that particularly CGRU are the most promising ANN models for long-time quantum dynamics simulations.

All ANN models are however less accurate than KRR methods with nonlinear kernels and take more time to train and predict. Often quoted poor computational scaling of KRR with the increasing system size may pose some technical problems for large data sets which may be needed for large systems, longer and larger number of training trajectories. Nevertheless, many approaches have been suggested to mitigate this problem [172–177] and we are also working on implementation of alternative approaches for large data sets.

One can reasonably anticipate that our results might depend on the strategy used to compare ML models based on ANNs. Here we chose to build and compare ANN models with approximately the same number of trainable parameters. There is however no proven best way to compare the performance of ANNs with fundamentally different types of layers. We believe that the strategy chosen in this work should provide a faithful comparison of ANN models. Additionally, our conclusion advocating for KRR methods holds irrespective of the strategy used to compare ANN models because KRR models studied in this work, are both faster and more accurate than ANN models. Addition of more layers to ANN models will likely make them more accurate (although overfitting should be carefully checked in this case) but it will necessarily make such models more computationally expensive.

FFNN may seem as a reasonable compromise between accuracy and computational cost. However, one needs to bear in mind that FFNN models require a fixed-size input. Of course an input to FFNN models can be padded with zeros, but it necessarily alters the representation of the underlying physics in the input data and, therefore, the performance of such models is not expected to be strong.

It is important to note that the actual performance of a computational method depends on many factors such as algorithmic and implementation details, program optimization, parallelization, hardware, and on the computational task itself. This makes benchmarking of ML methods implemented in different software packages difficult. Furthermore, many performance metrics can be chosen for benchmarking software efficiency such as floating-point operations, scaling, speedup, and efficiency of memory use. In this study, we focused on the clearest and most important metric which is time-to-solution measured as a wall-clock time for performing simulation for two tasks (quantum dynamics of symmetric and asymmetric spin-boson models) on the same machine and using state-of-the-art ANN and KRR implementations in KERAS (TENSORFLOW) and MLATOM, respectively. Despite the limitations of our approach and compromises associated with the comparison of many fundamentally different ML models, our results already uncover somewhat surprising underperformance of popular methods. Undoubtedly, more benchmarks are still needed, e.g. focusing on more complicated systems, different performance metrics, software, and hardware. Nonetheless this work begins to unravel the advantages and disadvantages of different ML methods in unexplored area of ML algorithms for quantum dissipative dynamics.

Finally, although many popular ML methods have been studied in the present article, the field of artificial intelligence and ML is growing rapidly making it nearly impossible to cover all recently developed algorithms. Our future work will focus on novel approaches to time-series modeling such as transformers [178]. Additionally, the CNN model employed in this work uses kernels of the same size. Even though the kernel size was optimized for the given task, the use of inception modules [179] that include multiple filters of varying size might improve the performance of CNNs and will also be tested.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://doi.org/10.6084/m9.figshare.15134649; https://github.com/kananenka-group/spin-boson_ML_benchmark.

Acknowledgments

A A K acknowledges the Ralph E Powe Junior Faculty Enhancement Award from Oak Ridge Associated Universities. This work was also supported by the startup funds of the College of Arts and Sciences and the Department of Physics and Astronomy of the University of Delaware. P O D acknowledges funding by the National Natural Science Foundation of China (NSFC, No. 22003051), the Fundamental Research Funds for the Central Universities (No. 20720210092) and via the Lab project of the State Key Laboratory of Physical Chemistry of Solid Surfaces. K J R E acknowledges support by the Beyond Research Program between University of Delaware and Universidad Nacional de Colombia. Calculations were performed with high-performance computing resources provided by the University of Delaware and Xiamen University.

Conflict of interest

The authors have no conflicts to disclose.

Appendix A. Comparison of performance of initial and final KRR models

As obvious from table 3, initial and final KRR models are very close. Regression coefficients of the initial KRR models were fitted only on the sub-training set (the validation set was only used for hyperparameter optimization), while regression coefficients of the final models were optimized on the entire training set. By 'fitting' we mean analytical solution of the regression problem formulated for the respective set (training or sub-training). Note that both models were trained on the entire training set which includes the validation set (used directly for training in the final model and indirectly in the initial model). Fitting on the entire training set is a common practice in KRR model training with the rationale that it is desirable to include all points to ensure that no points are 'wasted'.

Table 3. Mean absolute prediction errors (MAEs), training, and average single step prediction times of all KRR models used in this work. KRR-L, KRR-G, KRR-DP, KRR-E, and KRR-M*n* ($n = 1, 2, 3, 4$) denote kernel ridge regression models with linear kernel, Gaussian kernel, decaying-periodic kernel, exponential kernel, and Matérn kernel with $n = 1, 2, 3, 4$, respectively. The MAEs are shown for both symmetric and asymmetric spin-boson models. The regression coefficients of the final and initial KRR models were respectively fitted on 100% training set and sub-training set (80% of training set, i.e. the same approach that was used for training of all ANN models).

	Mean Absolute Prediction Error	
Model	Symmetric (initial | final)	Asymmetric (initial | final)
KRR-L	$(1.2 \mid 1.2)\times 10^{-2}$	$(6.5 \mid 5.8)\times 10^{-2}$
KRR-G	$(4.7 \mid 3.8)\times 10^{-4}$	$(1.2 \mid 1.3)\times 10^{-3}$
KRR-DP	$(4.3 \mid 4.5)\times 10^{-4}$	$(2.0 \mid 2.2)\times 10^{-3}$
KRR-E	$(2.1 \mid 2.5)\times 10^{-3}$	$(3.3 \mid 3.9)\times 10^{-3}$
KRR-M1	$(2.4 \mid 2.0)\times 10^{-4}$	$(1.3 \mid 1.6)\times 10^{-3}$
KRR-M2	$(2.2 \mid 3.2)\times 10^{-4}$	$(2.7 \mid 2.6)\times 10^{-3}$
KRR-M3	$(2.0 \mid 2.3)\times 10^{-4}$	$(2.3 \mid 2.1)\times 10^{-3}$
KRR-M4	$(2.3 \mid 2.4)\times 10^{-4}$	$(2.1 \mid 2.0)\times 10^{-3}$

Appendix B. Particle swarm optimization

The hyperparameter optimization of the 1D CNN model was performed using the heuristic particle swarm optimization (PSO) algorithm [180]. The PSO algorithm was inspired by the observation of the motion of swarms of birds and insects, where each swarm member is guided not only by the best solution for itself but also by the best solution seen by the entire population.

An object of a swarm that moves around in the search space is called a particle. A new update of the swarm is called a new generation. The particle in the swarm includes variables (hyperparameters of the model) updated during the optimization based on the information about the previous best (test MAE) states of the particle and the swarm itself. Each variable of an individual particle has velocity $\mathbf{v}(t)$, position $\mathbf{x}(t)$, and the best particle's position $\mathbf{x}_\mathrm{p}(t)$ which has generated the smallest MAE during the coarse of its trajectory. Additionally, there is a global variable that contains the best global position $\mathbf{x}_\mathrm{g}(t)$ of the swarm. Each generation the position and velocity of a particle are updated according to

$$\begin{aligned}\mathbf{v}_i(t+1) &= w\mathbf{v}_i(t) + c_\mathrm{p}\left(\mathbf{x}_{\mathrm{p}_i}(t) - \mathbf{x}_i(t)\right) + c_\mathrm{g}\left(\mathbf{x}_{\mathrm{g}_i}(t) - \mathbf{x}_i(t)\right)\\ \mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t),\end{aligned} \tag{B1}$$

where c_p, c_g, and w are cognitive, social, and inertia coefficients. These coefficients quantify how much the particle is directed toward the best solution seen by itself, by the swarm, and in the previous direction.

In the present work, the search space is conformed by the space of hyperparameters of the 1D CNN model as described in section 3.2.2. The values of the parameters are restricted to be positive at each step. If a parameter takes a negative value it is replaced by a random number in an interval of values initially set for the parameter. The following values of the coefficients are used $w = 0.729$, $c_\mathrm{p} = 1.49445$, $c_\mathrm{g} = 1.49445$.

ORCID iDs

Luis E Herrera Rodríguez https://orcid.org/0000-0002-5061-8285
Arif Ullah https://orcid.org/0000-0003-1702-3463
Kennet J Rueda Espinosa https://orcid.org/0000-0002-8828-1040
Pavlo O Dral https://orcid.org/0000-0002-2975-9876
Alexei A Kananenka https://orcid.org/0000-0001-9422-7203

References

[1] Weiss U 2012 *Quantum Dissipative Systems* (Singapore: World Scientific)
[2] Breuer H-P and Petruccione F 2002 *The Theory of Open Quantum Systems* (New York, NY: Oxford University Press)
[3] Leggett A J, Chakravarty S, Dorsey A T, Fisher M P A, Garg A and Zwerger W 1987 *Rev. Mod. Phys.* **59** 1
[4] Tanimura Y and Kubo R 1989 *J. Phys. Soc. Japan* **58** 1199
[5] Tanimura Y 2020 *J. Chem. Phys.* **153** 020901
[6] Meyer H-D, Manthe U and Cederbaum L 1990 *Chem. Phys. Lett.* **165** 73
[7] Wang H and Thoss M 2003 *J. Chem. Phys.* **119** 1289
[8] Makarov D E and Makri N 1994 *Chem. Phys. Lett.* **221** 482
[9] Prior J, Chin A W, Huelga S F and Plenio M B 2010 *Phys. Rev. Lett.* **105** 050404
[10] Ren J, Shuai Z and Kin-Lic Chan G 2018 *J. Chem. Theory Comput.* **14** 5027
[11] Luo B, Ye J, Guan C and Zhao Y 2010 *Phys. Chem. Chem. Phys.* **12** 15073

[12] Cohen G and Rabani E 2011 *Phys. Rev.* B **84** 075150
[13] Cohen G, Gull E, Reichman D R and Millis A J 2015 *Phys. Rev. Lett.* **115** 266802
[14] Greene S M and Batista V S 2017 *J. Chem. Theory Comput.* **13** 4034
[15] Yan Y-A and Shao J 2016 *Front. Phys.* **11** 1
[16] Hsieh C-Y and Cao J 2018a *J. Chem. Phys.* **148** 014103
[17] Hsieh C-Y and Cao J 2018b *J. Chem. Phys.* **148** 014104
[18] Han L, Ullah A, Yan Y-A, Zheng X, Yan Y and Chernyak V 2020 *J. Chem. Phys.* **152** 204105
[19] Ullah A, Han L, Yan Y-A, Zheng X, Yan Y and Chernyak V 2020 *J. Chem. Phys.* **152** 204106
[20] Nakajima S 1958 *Prog. Theor. Phys.* **20** 948
[21] Zwanzig R 1960 *J. Chem. Phys.* **33** 1338
[22] Shi Q and Geva E 2003 *J. Chem. Phys.* **119** 12063
[23] Kelly A and Markland T E 2013 *J. Chem. Phys.* **139** 014104
[24] Mulvihill E and Geva E 2021 *J. Phys. Chem.* B **125** 9834
[25] Mulvihill E, Lenn K M, Gao X, Schubert A, Dunietz B D and Geva E 2021 *J. Chem. Phys.* **154** 204109
[26] Brian D and Sun X 2021 *Chin. J. Chem. Phys.* **34** 497
[27] Cerrillo J and Cao J 2014 *Phys. Rev. Lett.* **112** 110401
[28] Kananenka A A, Hsieh C-Y, Cao J and Geva E 2016 *J. Phys. Chem. Lett.* **7** 4809
[29] Buser M, Cerrillo J, Schaller G and Cao J 2017 *Phys. Rev.* A **96** 062122
[30] Gelzinis A, Rybakovas E and Valkunas L 2017 *J. Chem. Phys.* **147** 234108
[31] Chen Y-Q, Ma K-L, Zheng Y-C, Allcock J, Zhang S and Hsieh C-Y 2020 *Phys. Rev. Appl.* **13** 034045
[32] Herrera Rodríguez L E and Kananenka A A 2021 *J. Phys. Chem. Lett.* **12** 2476
[33] Ullah A and Dral P O 2021 *New J. Phys.* **23** 113019
[34] Ullah A and Dral P O 2022 *Nat. Commun.* **13** 1930
[35] Ullah A and Dral P O 2022 *J. Phys. Chem. Lett.* **13** 6037
[36] Box G, Jenkins G, Reinsel G and Ljung G 2015 *Time Series Analysis: Forecasting and Control* (*Wiley Series in Probability and Statistics*) (New York: Wiley)
[37] Ariyo A A, Adewumi A O and Ayo C K 2014 *2014 UKSim-AMSS 16th Int. Conf. on Computer Modelling and Simulation* pp 106–12
[38] Khashei M and Bijari M 2011 *Appl. Soft Comput.* **11** 2664
[39] Schmidhuber J 2015 *Neural Netw.* **61** 85
[40] Goodfellow I, Bengio Y and Courville A 2016 *Deep Learning* (*Adaptive Computation and Machine Learning Series*) (Cambridge, MA: MIT Press)
[41] Sherstinsky A 2020 *Physica* D **404** 132306
[42] Hochreiter S and Schmidhuber J 1997 *Neural Comput.* **9** 1735
[43] Dral P O, Kananenka A A, Ge F and Xue B-X 2023 Neural networks *Quantum Chemistry in the Age of Machine Learning* ed Dral P O (Amsterdam:Elsevier) ch 8, pp 183–204
[44] Ye Q, Szeto W Y and Wong S C 2012 *IEEE Trans. Intell. Transp. Syst.* **13** 1727
[45] Siami-Namini S, Tavakoli N and Siami Namin A 2018 *2018 17th IEEE Int. Conf. on Machine Learning and Applications* (*ICMLA*) pp 1394–401
[46] Siami-Namini S, Tavakoli N and Namin A S 2019 *2019 IEEE Int. Conf. on Big Data* (*Big Data*) pp 3285–92
[47] Malhotra P, Tv V, Vig L, Agarwal P and Shroff G 2017 arXiv:1706.08838
[48] Fawaz H I, Forestier G, Weber J, Idoumghar L and Muller P-A 2019 *Data Min. Knowl. Discov.* **33** 917
[49] Kashiparekh K, Narwariya J, Malhotra P, Vig L and Shroff G 2019 *2019 Int. Joint Conf. on Neural Networks (IJCNN)* (IEEE) pp 1–8
[50] Moghar A and Hamiche M 2020 *Proc. Comput. Sci.* **170** 1168
[51] Fischer T and Krauss C 2018 *Eur. J. Oper. Res.* **270** 654
[52] Kim J and Moon N 2019 *J. Ambient Intell. Humaniz. Comput.* (https://doi.org/10.1007/s12652-019-01398-9)
[53] Kumar D, Sarangi P K and Verma R 2021 *Mater. Today Proc.*
[54] Graves A 2012 *Supervised Sequence Labelling With Recurrent Neural Networks* (*Studies in Computational Intelligence*) (Berlin: Springer)
[55] Graves A and Schmidhuber J 2005 *Neural Netw.* **18** 602 iJCNN 2005
[56] Graves A 2012b arXiv:1211.3711
[57] Graves A, Mohamed A-r and Hinton G 2013 *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (IEEE) pp 6645–9
[58] Hannun A *et al* 2014 arXiv:1412.5567
[59] Xiong W, Droppo J, Huang X, Seide F, Seltzer M, Stolcke A, Yu D and Zweig G 2016 arXiv:1610.05256
[60] Graves A, Fernández S, Liwicki M, Bunke H and Schmidhuber J 2007 *Proc. 20th Int. Conf. on Neural Information Processing Systems* (*NIPS'07*) (Red Hook, NY: Curran Associates Inc.) pp 577–84
[61] Graves A and Schmidhuber J 2009 *Advances in Neural Information Processing Systems* vol 21, ed D Koller, D Schuurmans, Y Bengio and L Bottou (Curran Associates, Inc.
[62] Tang Y, Huang Y, Wu Z, Meng H, Xu M and Cai L 2016 *2016 IEEE Int. Conf. on Acoustics, Speech and Signal Processing* (*ICASSP*) pp 6125–9
[63] Yin W, Kann K, Yu M and Schütze H 2017 arXiv:1702.01923
[64] Sutskever I, Vinyals O and Le Q V 2014 *Adv. Neural Inf. Process. Syst.* **27** 3104–12
[65] Bahdanau D, Cho K and Bengio Y 2014 arXiv:1409.0473
[66] Shido Y, Kobayashi Y, Yamamoto A, Miyamoto A and Matsumura T 2019 *Int. Joint Conf. on Neural Networks* (*IJCNN*) pp 1–8
[67] Choi E, Bahadori M T, Song L, Stewart W F and Sun J 2017 arXiv:1611.07012v3
[68] Lynn H M, Pan S B and Kim P 2019 *IEEE Access* **7** 145395
[69] Gupta P, Malhotra P, Narwariya J, Vig L and Shroff G 2021 arXiv:1904.00655
[70] Ma F, Chitta R, Zhou J, You Q, Sun T and Gao J 2017 *Proc. 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* pp 1903–11
[71] van Lint J, Hoogendoorn S and van Zuylen H 2005 *Transp. Res.* C **13** 347
[72] Zhao Z, Chen W, Wu X, Chen P C Y and Liu J 2017 *IET Intell. Transp. Syst.* **11** 68
[73] Cui Z, Ke R, Pu Z and Wang Y 2019 arXiv:1801.02143

[74] Eck D and Schmidhuber J 2002 A first look at music composition using lstm recurrent neural networks (Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale)
[75] Boulanger-Lewandowski N, Bengio Y and Vincent P 2012 arXiv:1206.6392
[76] Srivastava N, Mansimov E and Salakhudinov R 2015 *Proc. 32nd Int. Conf. on Machine Learning* vol 37, ed F Bach and D Blei (Lille: PMLR) pp 843–52
[77] Habi H V and Messer H 2019 *2019 IEEE Int. Workshop on Signal Processing Systems (SiPS)* pp 184–8
[78] Gupta A, Müller A T, Huisman B J H, Fuchs J A, Schneider P and Schneider G 2018 *Mol. Inform.* **37** 1700111
[79] Segler M H S, Kogej T, Tyrchan C and Waller M P 2018 *ACS Cent. Sci.* **4** 120
[80] Wei W and Yan X 2019 *IOP Conf. Ser.: Mater. Sci. Eng.* **688** 033022
[81] Abbasimehr H, Shabani M and Yousefi M 2020 *Comput. Ind. Eng.* **143** 106435
[82] Zhang L, Xu C, Gao Y, Han Y, Du X and Tian Z 2020 *Tsinghua Sci. Technol.* **25** 712
[83] Lyu H, Lu H and Mou L 2016 *Remote Sens.* **8** 506
[84] Ienco D, Gaetano R, Dupaquier C and Maurel P 2017 *IEEE Geosci. Remote Sens. Lett.* **14** 1685
[85] Bhoopchand A, Rocktäschel T, Barr E and Riedel S 2016 arXiv:1611.08307
[86] Pineda F J 1987 *Phys. Rev. Lett.* **59** 2229
[87] Gers F A, Schmidhuber J and Cummins F 2000 *Neural Comput.* **12** 2451
[88] Gers F, Schmidhuber J and Cummins F 1999 *1999 9th Int. Conf. on Artificial Neural Networks (ICANN 99) (Conf. Publ. No. 470)* vol 2 pp 850–5
[89] Cho K, Van Merriënboer B, Bahdanau D, and Y Bengio 2014 arXiv:1409.1259
[90] Chung J, Cho K, and Bengio Y 2016 arXiv:1603.06147
[91] Jozefowicz R, Zaremba W and Sutskever I 2015 *Proc. 32nd Int. Conf. on Machine Learning* vol 37, ed F Bach and D Blei (Lille: PMLR) pp 2342–50
[92] Schuster M and Paliwal K 1997 *IEEE Trans. Signal Process.* **45** 2673
[93] Baldi P, Brunak S, Frasconi P, Soda G and Pollastri G 1999 *Bioinformatics* **15** 937
[94] LeCun Y, Boser B, Denker J S, Henderson D, Howard R E, Hubbard W and Jackel L D 1989 *Neural Comput.* **1** 541
[95] Wang Z, Yan W and Oates T 2016 arXiv:1611.06455
[96] Cui Z, Chen W and Chen Y 2016 arXiv:1603.06995
[97] Serrà J, Pascual S and Karatzoglou A 2018 arXiv:1805.03908
[98] Zheng Y, Liu Q, Chen E, Ge Y and Zhao J L 2014 *Web-Age Information Management*, ed F Li, G Li, S-W Hwang, B Yao and Z Zhang (Cham: Springer) pp 298–310
[99] Zheng Y, Liu Q, Chen E, Ge Y and Zhao J L 2016 *Front. Comput. Sci.* **10** 96
[100] Interdonato R, Ienco D, Gaetano R and Ose K 2018 arXiv:1809.07589
[101] Ismail Fawaz H, Lucas B, Forestier G, Pelletier C, Schmidt D F, Weber J, Webb G I, Idoumghar L, Muller P-A and Petitjean F 2020 *Data Min. Knowl. Discov.* **34** 1936–62
[102] Tang W, Long G, Liu L, Zhou T, Jiang J, and Blumenstein M 2021 arXiv:2002.10061
[103] Rajpurkar P, Hannun A Y, Haghpanahi M, Bourn C, and Ng A Y 2017 arXiv:1707.01836
[104] Roy S, Kiral-Kornek I, and Harrer S2018 arXiv:1802.00308
[105] Schirrmeister R T, Gemein L, Eggensperger K, Hutter F, and Ball T 2018 arXiv:1708.08012
[106] Sercu T, Puhrsch C, Kingsbury B, and LeCun Y 2016 arXiv:1509.08967
[107] Choi K, Fazekas G, Sandler M, and Cho K 2016 arXiv:1609.04243
[108] You S D, Liu C-H and Chen W-K 2018 *Hum.-Centric Comput. Inf. Sci.* **8** 34
[109] Karim F, Majumdar S, Darabi H and Chen S 2018 *IEEE Access* **6** 1662–9
[110] Sheykhivand S, Mousavi Z, Rezaii T Y and Farzamnia A 2020 *IEEE Access* **8** 139332
[111] Eapen J, Bein D and Verma A 2019 *2019 IEEE 9th Annual Computing and Communication Workshop and Conf. (CCWC)* pp 0264–70
[112] Shi X, Chen Z, Wang H, Yeung D-Y, kin Wong W, and chun Woo W 2015 arXiv:1506.04214
[113] Müller K R, Smola A J, Rätsch G, Schölkopf B, Kohlmorgen J, and Vapnik V 1997 *Artificial Neural Networks (ICANN 1997)* ed W Gerstner, A Germond, M Hasler and J D Nicoud (Berlin: Springer) pp 999–1004
[114] Sapankevych N I and Sankar R 2009 *IEEE Comput. Intell. Mag.* **4** 24
[115] Haworth J, Shawe-Taylor J, Cheng T and Wang J 2014 *Transp. Res.* C **46** 151
[116] Pinheiro M Jr and Dral P O 2023 Kernel methods *Quantum Chemistry in the Age of Machine Learning* ed P O Dral (Amsterdam: Elsevier) ch 9, pp 205–32
[117] Gneiting T, Kleiber W and Schlather M 2010 *J. Am. Stat. Assoc.* **105** 1167
[118] Cuturi M and Doucet A 2011 arXiv:1101.0673
[119] Akimov A V 2021 *J. Phys. Chem. Lett.* **12** 12119
[120] Secor M, Soudackov A V and Hammes-Schiffer S 2021 *J. Phys. Chem. Lett.* **12** 10654
[121] Yang B, He B, Wan J, Kubal S and Zhao Y 2020 *Chem. Phys.* **528** 110509
[122] Bandyopadhyay S, Huang Z, Sun K and Zhao Y 2018 *Chem. Phys.* **515** 272
[123] Banchi L, Grant E, Rocchetto A and Severini S 2018 *New J. Phys.* **20** 123030
[124] Wu D, Hu Z, Li J and Sun X 2021 *J. Chem. Phys.* **155** 224104
[125] Lin K, Peng J, Gu F L and Lan Z 2021 *J. Phys. Chem. Lett.* **12** 10225
[126] Choi M, Flam-Shepherd D, Kyaw T H and Aspuru-Guzik A 2022 *Phys. Rev.* A **105** 042403
[127] Tsai S-T, Fields E, Xu Y, Kuo E-J, and Tiwary P 2022 arXiv:2203.00597
[128] Tang D, Jia L, Shen L, and Fang W-H 2022 arXiv:2206.13780
[129] Dral P O and Barbatti M 2021 *Nat. Rev. Chem.* **5** 388–405
[130] Westermayr J and Marquetand P 2020 *Mach. Learn.: Sci. Technol.* **1** 043001
[131] Westermayr J and Marquetand P 2021 *Chem. Rev.* **121** 9873–926
[132] Lin K, Peng J, Xu C, Gu F L, and Lan Z 2022a arXiv:2205.03600
[133] Lin K, Peng J, Xu C, Gu F L, and Lan Z 2022b arXiv:2207.05556
[134] Makhlin Y, Schön G and Shnirman A 2001 *Rev. Mod. Phys.* **73** 357
[135] Alvermann A and Fehske H 2009 *Phys. Rev. Lett.* **102** 150601
[136] Winter A, Rieger H, Vojta M and Bulla R 2009 *Phys. Rev. Lett.* **102** 030601
[137] Garg A, Onuchic J N and Ambegaokar V 1985 *J. Chem. Phys.* **83** 4491

[138] Wang H, Song X, Chandler D and Miller W H 1999 *J. Chem. Phys.* **110** 4828
[139] Hornik K, Stinchcombe M and White H 1989 *Neural Netw.* **2** 359
[140] Cybenko G 1989 *Math. Control Signals Syst.* **2** 303
[141] Leshno M, Lin V Y, Pinkus A and Schocken S 1993 *Neural Netw.* **6** 861
[142] Zhou Y-T and Chellappa R 1988 *IEEE 1988 Int. Conf. on Neural Networks* vol 2 pp 71–78
[143] Rumelhart D E, Hinton G E and Williams R J 1986 *Nature* **323** 533
[144] Schäfer A M and Zimmermann H G 2006 *Artificial Neural Networks – ICANN 2006* ed S D Kollias, A Stafylopatis, W Duch and E Oja (Berlin: Springer) pp 632–40
[145] Chollet F *et al* 2015 Keras (available at: https://github.com/fchollet/keras)
[146] Hochreiter J 1991 Untersuchungen zu dynamischen neuronalen netzen *PhD Thesis* technische Universitat Munchen
[147] Bengio Y, Simard P and Frasconi P 1994 *IEEE Trans. Neural Netw.* **5** 157
[148] Bengio Y, Frasconi P and Simard P Y 1993 *Proc. of Int. Conf. on Neural Networks* (*ICNN 1988*) (*San Francisco, CA, USA, 28 March–1 April 1993*) (IEEE) pp 1183–8
[149] Pascanu R, Mikolov T, and Bengio Y 2013 arXiv:1211.5063
[150] Sussillo D and Abbott L 2014 arXiv:1412.6558
[151] Chung J, Gulcehre C, Cho K and Bengio Y 2015 *Int. Conference on Machine Learning* (PMLR) pp 2067–75
[152] Abadi M *et al* 2015 TensorFlow: large-scale machine learning on heterogeneous systems *Software* (tensorflow.org)
[153] Stulp F and Sigaud O 2015 *Neural Netw.* **69** 60
[154] Dral P O 2019 *J. Comput. Chem.* **40** 2339
[155] Hastie T, Tibshirani R, Friedman J H and Friedman J H 2009 *The Elements of Statistical Learning: Data Mining, Inference and Prediction* vol 2 (Berlin: Springer)
[156] Rasmussen C and Williams C 2005 *Gaussian Processes for Machine Learning* (Adaptive Computation and Machine Learning series (MIT Press)
[157] Dral P O, Ge F, Xue B-X, Hou Y-F, Pinheiro M, Huang J and Barbatti M 2021 *Top. Curr. Chem.* **379** 1
[158] Dral P O, Zheng P, Xue B-X, Ge F, Hou Y-F and Pinheiro Jr M 2013–2022 *Mlatom: a Package for Atomistic Simulations With Machine Learning, Development Version* (Xiamen: Xiamen University) (available at: http://MLatom.com) (Accessed 29 June 2022)
[159] Pedregosa F *et al* 2011 *J. Mach. Learn. Res.* **12** 2825
[160] Johansson J, Nation P and Nori F 2012 *Comput. Phys. Commun.* **183** 1760
[161] Kingma D P and Ba J 2017 arXiv:1412.6980
[162] Glorot X and Bengio Y 2010 *Proc. 13th Int. Conf. on Artificial Intelligence and*, vol 9, ed Y W Teh and M Titterington (Chia Laguna Resort, Sardinia: PMLR) pp 249–56
[163] Bergstra J, Bardenet R, Bengio Y and Kégl B 2011 Algorithms for hyper-parameter optimization *Advances in Neural Information Processing Systems* vol 24, ed J Shawe-Taylor, R Zemel, P Bartlett, F Pereira and K Q Weinberger (Curran Associates, Inc.)
[164] Bergstra J, Komer B, Eliasmith C, Yamins D L K and Cox D D 2015 *Comput. Sci. Discov.* **8** 014008
[165] Stérin T, Farrugia N and Gripon V 2017 *COGNITIVE 2017: Ninth Int. Conf. on Advanced Cognitive Technologies and Applications* (*Greece*, *Athènes*) pp 76–81
[166] Gupta G and Saini S 2020 *Mach. Learn.: Sci. Technol* **1** 025013
[167] Häse F, Kreisbeck C and Aspuru-Guzik A 2017 *Chem. Sci.* **8** 8419–26
[168] Hansen K, Montavon G, Biegler F, Fazli S, Rupp M, Scheffler M, von Lilienfeld O A, Tkatchenko A and Müller K-R 2013 *J. Chem. Theory Comput.* **9** 3404
[169] Kamath A, Vargas-Hernández R A, Krems R V, Carrington Jr T and Manzhos S 2018 *J. Chem. Phys.* **148** 241702
[170] Nguyen T T, Szekely E, Imbalzano G, Behler J, Csányi G, Ceriotti M, Gotz A W and Paesani F 2018 *J. Chem. Phys.* **148** 241725
[171] Pinheiro Jr M, Ge F, Ferré N, Dral P O and Barbatti M 2021 *Chem. Sci.* **12** 14396
[172] Snelson E and Ghahramani Z 2005 *Adv. Neural Inf. Process. Syst.* **18**
[173] Deringer V L, Bartók A P, Bernstein N, Wilkins D M, Ceriotti M and Csányi G 2021 *Chem. Rev.* **121** 10073
[174] Rahimi A and Recht B 2007 *Adv. Neural Inf. Process. Syst.* **20**
[175] Yu F X X, Suresh A T, Choromanski K M, Holtmann-Rice D N and Kumar S 2016 *Adv. Neural Inf. Process. Syst.* **29**
[176] Hu D, Xie Y, Li X, Li L and Lan Z 2018 *J. Phys. Chem. Lett.* **9** 2725
[177] Browning N J, Faber F A and Anatole von Lilienfeld O 2022 arXiv:2206.01580v2
[178] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser Ł and Polosukhin I 2017 *Adv. Neural Inf. Process. Syst.* **30**
[179] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V and Rabinovich A 2015 *2015 IEEE Conf. on Computer Vision and Pattern Recognition* (*CVPR*) pp 1–9
[180] Kennedy J and Eberhart R 1995 *Proc. ICNN'95 - Int. Conf. on Neural Networks* vol 4 pp 1942–8