Journal of Data Science 23 (4), 676–694
October 2025

DOI: 10.6339/25-JDS1167
Data Science Reviews

A Statistician's Selective Review of Neural Network Modeling: Algorithms and Applications

Chunming Zhang[1,*], Zhengjun Zhang[2,1], Xinrui Zhong[1], Jialuo Li[1], and Zhihao Zhao[1]

[1]*University of Wisconsin-Madison, Department of Statistics, Madison, Wisconsin, U.S.A.*
[2]*School of Economics and Management, and MOE Social Science Laboratory of Digital Economic Forecasts and Policy Simulation, University of Chinese Academy of Sciences, AMSS Center for Forecasting Sciences, Chinese Academy of Sciences, Beijing, China*

Abstract

Deep neural networks have a wide range of applications in data science. This paper reviews neural network modeling algorithms and their applications in both supervised and unsupervised learning. Key examples include: (i) binary classification and (ii) nonparametric regression function estimation, both implemented with feedforward neural networks (FNN); (iii) sequential data prediction using long short-term memory (LSTM) networks; and (iv) image classification using convolutional neural networks (CNN). All implementations are provided in MATLAB, making these methods accessible to statisticians and data scientists to support learning and practical application.

Keywords *classification; nonparametric regression; prediction; time series*

1 Introduction

Neural networks (NN) have become a cornerstone of modern data science, offering flexible and powerful tools for a wide range of applications, from image classification to natural language processing (Alzubaidi et al., 2021). Inspired by biological neural systems, artificial neural networks (ANNs) have evolved significantly over recent decades. In particular, the development of deep learning techniques has revolutionized the field, enabling neural networks to automatically learn hierarchical representations from complex data. For a comprehensive overview of the state of the art in deep learning, see Goodfellow et al. (2016), and for an in-depth review aimed at applied mathematicians, refer to Higham and Higham (2019).

Statisticians have long been interested in models capable of capturing nonlinear relationships and interactions within data. Neural networks, particularly deep neural networks (DNNs), offer a nonparametric framework for modeling these complexities without relying on strict parametric assumptions. While traditional statistical models often emphasize interpretability and inference, neural networks excel in prediction and pattern recognition in high-dimensional settings. For theoretical insights, see Schmidt-Hieber (2020); Farrell et al. (2021). For statisticians looking to incorporate neural networks into their toolkit, understanding the benefits and limitations of these models is essential.

This paper aims to bridge the gap between traditional statistical approaches and contempo-

*Corresponding author. Email: czhang3@wisc.edu.

Received October 25, 2024; Accepted January 8, 2025

rary machine learning techniques by providing a statistician-friendly, selective review of neural network modeling algorithms. A broader review of machine learning can be found in Jordan and Mitchell (2015). We begin with an overview of the core building blocks of neural networks, including activation functions, feedforward neural network architectures for regression and classification tasks, and optimization methods. We then examine more advanced architectures, such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs), which have proven effective for sequential and image data, respectively.

The practical applications of neural networks are demonstrated through four illustrative examples: (i) feedforward NN for classification, (ii) feedforward NN for nonparametric function estimation, (iii) long short-term memory (LSTM) networks for sequence prediction, and (iv) convolutional neural networks (CNNs) for image classification. All examples are implemented in MATLAB to encourage hands-on learning and facilitate the broader adoption of neural network methods among statisticians and data scientists.

By exploring both the learning aspects and practical applications, this paper aims to provide statisticians with the knowledge needed to understand, implement, and critically assess neural network models in their work. For additional research topics on neural networks, see Kramer (1991) for nonlinear principal component analysis (PCA) of data vectors; Zhong et al. (2024) and related references for nonlinear dimension reduction and curve reconstruction of functional data; and Katthi et al. (2020) for canonical correlation analysis.

The rest of the paper is organized as follows: Section 2 describes the basic procedure for artificial neural network modeling when the input feature is either a scalar or a vector. Feedforward neural networks are illustrated in Section 3 for classification tasks and in Section 4 for nonparametric regression function estimation. Section 5 discusses LSTM models for time series prediction, and Section 6 covers CNN models for image data. Technical details and numerical illustrations are provided in Appendices A and B of the supplementary file.

2 Deep Neural Network Modeling

A *deep neural network* (DNN) is a broad term for any neural network with multiple layers (typically more than two), while a *feedforward neural network* (FNN) specifically refers to a neural network architecture (Ripley, 1996) in which connections between nodes do not form cycles with applications in Sections 3 and 4. DNNs also encompass other architectures like *recurrent neural networks* (RNNs) in Section 5, and *convolutional neural networks* (CNNs) in Section 6, each tailored to different types of data and applications.

For clarity of presentation, we begin by introducing some necessary notations. Commonly used nonlinear activation functions $\sigma(z)$, for a scalar z, include:

- **The sigmoid function**:

$$\text{sigmoid}(z) = 1/\{1 + \exp(-z)\} \in [0, 1]. \tag{1}$$

- **The tanh function** (Vogl et al., 1988, short for hyperbolic tangent) is a mathematical function that maps real-valued numbers to the interval $[-1, 1]$:

$$\tanh(z) = 2 \cdot \text{sigmoid}(2z) - 1 \in [-1, 1]. \tag{2}$$

- **The ReLU (Rectified Linear Unit) function** is convex:

$$\text{ReLU}(z) = \max(z, 0) = \begin{cases} z, & \text{if } z \geqslant 0, \\ 0, & \text{if } z \leqslant 0 \end{cases} \in (0, \infty). \tag{3}$$

For a column vector $\boldsymbol{z} = (z_1, \ldots, z_m)^\top \in \mathbb{R}^m$, define the vector

$$\sigma(\boldsymbol{z}) = (\sigma(z_1), \ldots, \sigma(z_m))^\top \in \mathbb{R}^m,$$

which applies to each component z_j of $\boldsymbol{z}$. The case for a row vector $\boldsymbol{z}$ is defined similarly.

The softmax function is commonly used as the activation function in the output layer of a neural network for multi-class classification tasks. Formally, the standard (unit) softmax function of $\boldsymbol{z} = (z_1, \ldots, z_K)^\top$, where $K \geqslant 2$, is defined as:

$$\text{softmax}(\boldsymbol{z}) = \Big(\frac{\exp(z_1)}{\sum_{k=1}^K \exp(z_k)}, \ldots, \frac{\exp(z_K)}{\sum_{k=1}^K \exp(z_k)} \Big)^\top. \tag{4}$$

A comprehensive review of activation functions and their properties can be found in Zhang et al. (2024).

2.1 General Setup for Feedforward NN

The general setup for FNN modeling involves an architecture with $L > 2$ layers, where each Layer-ℓ contains $n^{[\ell]}$ nodes for $\ell \in \{1, \ldots, L\}$ (Hinton et al., 2006; Hinton, 2007). Given an input feature vector $\boldsymbol{X} \in \mathbb{R}^{n^{[1]}}$ and a set of parameters $\{(W^{[\ell]}, \boldsymbol{b}^{[\ell]}) : \ell = 2, \ldots, L\}$, which include weight matrices $W^{[\ell]} = (w_{j,k}^{[\ell]}) \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}}$ and bias vectors $\boldsymbol{b}^{[\ell]} = (b_1^{[\ell]}, \ldots, b_{n^{[\ell]}}^{[\ell]})^\top \in \mathbb{R}^{n^{[\ell]}}$, the forward-pass step computes the following quantities across the L layers in sequence:

$$\begin{array}{lll}
\text{Layer-1:} & \boldsymbol{a}^{[1]} \equiv \boldsymbol{a}^{[1]}(\boldsymbol{X}) = \boldsymbol{X} \in \mathbb{R}^{n^{[1]}}, & \text{(input layer);} \\
\text{Layer-2:} & \begin{cases} \boldsymbol{z}^{[2]} = W^{[2]}\, \boldsymbol{a}^{[1]} + \boldsymbol{b}^{[2]} \in \mathbb{R}^{n^{[2]}}, \\ \boldsymbol{a}^{[2]} \equiv \boldsymbol{a}^{[2]}(\boldsymbol{X}) = \sigma^{[2]}(\boldsymbol{z}^{[2]}) \in \mathbb{R}^{n^{[2]}}; \end{cases} & \\
\ldots & \ldots & \ldots \\
\text{Layer-}\ell\text{:} & \begin{cases} \boldsymbol{z}^{[\ell]} = W^{[\ell]}\, \boldsymbol{a}^{[\ell-1]} + \boldsymbol{b}^{[\ell]} \in \mathbb{R}^{n^{[\ell]}}, \\ \boldsymbol{a}^{[\ell]} \equiv \boldsymbol{a}^{[\ell]}(\boldsymbol{X}) = \sigma^{[\ell]}(\boldsymbol{z}^{[\ell]}) \in \mathbb{R}^{n^{[\ell]}}; \end{cases} & \text{for } \ell = 2, \ldots, L-1,\ \ L,
\end{array} \tag{5}$$

using the activation functions $\sigma^{[\ell]}(\cdot)$. In numerical implementations (e.g., MATLAB programming), the forward-pass scheme given in (5) can also be written as:

$$\underset{\text{original input}}{\underset{\text{Layer-1}}{\begin{cases} \boldsymbol{z}^{[1]} = [\,], \\ \boldsymbol{a}^{[1]} = \boldsymbol{X}; \end{cases}}} \ \ldots; \ \underset{\text{weighted input, activation}}{\underset{\text{Layer-}\ell}{\begin{cases} \boldsymbol{z}^{[\ell]} = W^{[\ell]}\, \boldsymbol{a}^{[\ell-1]} + \boldsymbol{b}^{[\ell]}, \\ \boldsymbol{a}^{[\ell]} = \sigma^{[\ell]}(\boldsymbol{z}^{[\ell]}); \end{cases}}} \ \ldots; \ \underset{\text{weighted input, output}}{\underset{\text{Layer-}L}{\begin{cases} \boldsymbol{z}^{[L]} = W^{[L]}\, \boldsymbol{a}^{[L-1]} + \boldsymbol{b}^{[L]}, \\ \boldsymbol{a}^{[L]} = \sigma^{[L]}(\boldsymbol{z}^{[L]}). \end{cases}}}$$

Particularly, for L layers in (5), the beginning $\ell = 1$ indicates the 'input layer', where there is no 'previous layer' and each node receives the input feature $\boldsymbol{X} \in \mathbb{R}^{n^{[1]}}$. On the other hand, the ending $\ell = L$ corresponds to the 'output layer', where there is no 'next layer' and these nodes provide the overall output. The intermediate layers $\ell \in \{2, \ldots, L-1\}$ correspond to the 'hidden layers'. See Figure 1 for an illustration of a 5-layer FNN. Refer to Farrell et al. (2021) for a theoretical study on the network depth, layer width, and the number of training samples.

Regarding model parametrization, the parameter $w_{j,k}^{[\ell]}$ in the weight matrix $W^{[\ell]}$ is the weight that node j at Layer-ℓ applies to the output from node k at Layer-$(\ell-1)$. The parameter

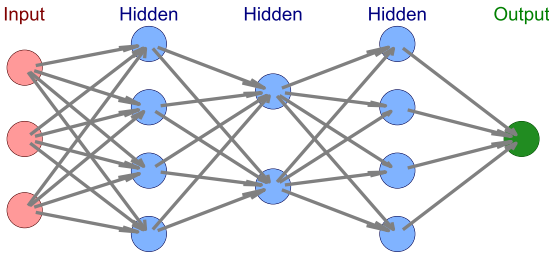


Figure 1: Illustration of an FNN with three hidden layers between the input and output layers. This is a fully connected network, where every node in one layer is connected to every node in the subsequent layer.

$b_j^{[\ell]}$ in the bias vector $\boldsymbol{b}^{[\ell]}$ is the bias term used by node j at Layer-ℓ. Accordingly, for layers $\ell \in \{2, \ldots, L-1, L\}$, the vector

$$\boldsymbol{z}^{[\ell]} = (z_1^{[\ell]}, \ldots, z_{n^{[\ell]}}^{[\ell]})^\top \in \mathbb{R}^{n^{[\ell]}}$$

consists of the weighted inputs:

$$z_j^{[\ell]} = \sum_{k=1}^{n^{[\ell-1]}} w_{j,k}^{[\ell]} a_k^{[\ell-1]} + b_j^{[\ell]}, \quad j = 1, \ldots, n^{[\ell]},$$

which are subsequently activated by the entries $a_j^{[\ell]} = \sigma^{[\ell]}(z_j^{[\ell]})$ in the vector

$$\boldsymbol{a}^{[\ell]} = (a_1^{[\ell]}, \ldots, a_{n^{[\ell]}}^{[\ell]})^\top \in \mathbb{R}^{n^{[\ell]}}.$$

Clearly, the parameter set $\boldsymbol{p} = \{W^{[2]}, \boldsymbol{b}^{[2]}; \ldots; W^{[L]}, \boldsymbol{b}^{[L]}\}$ contains a total of

$$\sum_{\ell=2}^{L} (n^{[\ell]} \times n^{[\ell-1]} + n^{[\ell]})$$

model parameters, which are learned from the training data and tested on the test data. During training, the neural network model is trained to minimize a scalar cost function:

$$\min_{\boldsymbol{p}} \text{Cost}(\boldsymbol{p}), \tag{6}$$

and the optimal solution of the model parameters is denoted by $\widehat{\boldsymbol{p}} = \{\widehat{W}^{[2]}, \widehat{\boldsymbol{b}}^{[2]}; \ldots; \widehat{W}^{[L]}, \widehat{\boldsymbol{b}}^{[L]}\}$.

In supervised learning, the training dataset is represented as $\{\boldsymbol{d}_i : i = 1, \ldots, N\}$, where each training sample $\boldsymbol{d}_i = (\boldsymbol{X}_i, \boldsymbol{Y}_i)$ consists of an input feature $\boldsymbol{X}_i$ and a target output $\boldsymbol{Y}_i$. This setup results in a predicted output $\boldsymbol{a}^{[L]}(\boldsymbol{X}_i)$, which can be more precisely expressed as $\boldsymbol{a}^{[L]}(\boldsymbol{X}_i; \boldsymbol{p})$ to emphasize its dependence on the parameter set $\boldsymbol{p}$. For example, in a regression setting, a commonly used cost function across all N training data points is:

$$\begin{aligned} \text{Cost}(\boldsymbol{p}) &= \text{Cost}(W^{[2]}, \boldsymbol{b}^{[2]}; \ldots; W^{[L]}, \boldsymbol{b}^{[L]}) \\ &= N^{-1} \sum_{i=1}^{N} 2^{-1} \|\boldsymbol{Y}_i - \boldsymbol{a}^{[L]}(\boldsymbol{X}_i; \boldsymbol{p})\|_2^2 = N^{-1} \sum_{i=1}^{N} C_{\boldsymbol{d}_i}^{[L]}(\boldsymbol{p}), \end{aligned} \tag{7}$$

with the quadratic loss function defined as:

$$C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p}) = 2^{-1} \|\boldsymbol{Y} - \boldsymbol{a}^{[L]}(\boldsymbol{X}; \boldsymbol{p})\|_2^2 \tag{8}$$

at an individual data point $\boldsymbol{d} = (\boldsymbol{X}, \boldsymbol{Y})$. As another example, in the case of two-class classification, for a given input feature $\boldsymbol{X}$ and a class label $Y_c \in \{1, 2\}$, the one-hot encoded true label vector can be defined as:

$$\boldsymbol{Y} = \begin{cases} (1, 0)^\top, & \text{if } Y_c = 1, \\ (0, 1)^\top, & \text{if } Y_c = 2. \end{cases} \tag{9}$$

Then, the quadratic loss function (8) still applies.

Remark 1 (Other loss functions)**.** Refer to Friedman et al. (2001); Zhang et al. (2023) for a discussion of other commonly used loss functions in regression and classification tasks. For example, cross-entropy is frequently applied to classification problems. Numerical illustrations for two-class classification are provided in Section 3, while multi-class classification examples are discussed in Section 6.3. In particular, Zhang et al. (2023) unifies commonly used loss functions (including the negative log-likelihood for generalized linear models from McCullagh and Nelder (1989)) using the Bregman Divergence (BD) framework and proposes robust variants (robust-BD) of Bregman Divergence.

For unsupervised learning with unlabeled data points $\boldsymbol{d}_i = \boldsymbol{X}_i$, procedures such as nonlinear dimension reduction via PCA are discussed in Kramer (1991).

2.2 Optimization Algorithms for Training Feedforward NN

To solve the optimization problem (6), commonly used algorithms include gradient descent (GD) and nonlinear least squares (**NLLS**, Coleman and Li (1994)), with **NLLS** being particularly designed for nonlinear least-squares problems. Here, we focus on describing the GD algorithm and its related variants.

The idea of the GD method for solving (6) is based on the first-order Taylor expansion of the cost function:

$$\begin{aligned} \text{Cost}(\boldsymbol{p} + \Delta_{\boldsymbol{p}}) &\approx \text{Cost}(\boldsymbol{p}) + \textstyle\sum_{r=1}^{s} \frac{\partial \text{Cost}(\boldsymbol{p})}{\partial p_r} \Delta_{p_r} \\ &= \text{Cost}(\boldsymbol{p}) + \{\nabla \text{Cost}(\boldsymbol{p})\}^\top \Delta_{\boldsymbol{p}}, \end{aligned} \tag{10}$$

where $\Delta_{\boldsymbol{p}} = (\Delta_{p_1}, \ldots, \Delta_{p_s})^\top$ (with s being the dimension of $\boldsymbol{p}$) is the direction vector, and the gradient vector is $\nabla \text{Cost}(\boldsymbol{p}) = (\frac{\partial \text{Cost}(\boldsymbol{p})}{\partial p_1}, \ldots, \frac{\partial \text{Cost}(\boldsymbol{p})}{\partial p_s})^\top$. To minimize the cost function $\text{Cost}(\boldsymbol{p})$ with respect to $\boldsymbol{p}$, the expression (10) suggests choosing the direction $\Delta_{\boldsymbol{p}}$ along the direction $-\nabla \text{Cost}(\boldsymbol{p})$, leading to the steepest descent method. In a single update step, the parameters are updated from $\boldsymbol{p}_1$ to $\boldsymbol{p}_2$ according to:

$$\boldsymbol{p}_2 = \boldsymbol{p}_1 - \eta \cdot \nabla \text{Cost}(\boldsymbol{p}_1), \tag{11}$$

where $\eta > 0$ is a small step size, also called the 'learning rate'. When the entire training dataset is used to compute the gradient of the cost function with respect to the model parameters at each iteration, the update (11) yields the batch GD (**BGD**):

$$\textbf{BGD}: \quad \boldsymbol{p}_2 = \boldsymbol{p}_1 - \eta \cdot \frac{1}{N} \textstyle\sum_{i=1}^{N} \nabla C_{\boldsymbol{d}_i}^{[L]}(\boldsymbol{p}_1). \tag{12}$$

However, this approach can be computationally expensive and slow, especially for large datasets, as it processes all data points before making a single update.

To simplify derivative computations in (12) at each iteration, various variants have been developed, including stochastic GD (**SGD**) and mini-batch GD (mini-**BGD**):

$$\textbf{SGD}: \quad \boldsymbol{p}_2 = \boldsymbol{p}_1 - \eta \cdot \nabla C_{\boldsymbol{d}_{k_i}}^{[L]}(\boldsymbol{p}_1), \quad i = 1, \ldots, N, \tag{13}$$

$$\text{mini-}\textbf{BGD}: \quad \boldsymbol{p}_2 = \boldsymbol{p}_1 - \eta \cdot \frac{1}{\#B_j} \sum_{i \in B_j} \nabla C_{\boldsymbol{d}_{k_i}}^{[L]}(\boldsymbol{p}_1), \quad j = 1, \ldots, N_{\mathrm{B}}. \tag{14}$$

Within each epoch in (13) and (14), $\{k_1, \ldots, k_N\}$ typically comes from a random permutation of $\{1, \ldots, N\}$, and N data points are split into N_{B} batches, each with batch size m_{B}.

The **Adam** (Adaptive Moment Estimation, Kingma and Ba (2015)) optimization algorithm, a GD method with adaptive learning rates for each parameter, can also be used as an alternative to **BGD** for updating neural network model parameters.

2.3 Backpropagation Step (for Explicitly Computing Partial Derivatives in Gradients)

As demonstrated in (8), the gradients $\nabla C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})$ of the loss functions $C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})$ from the output Layer-L are required for the optimization algorithm. This section discusses the backpropagation (short for 'backward propagation of errors') procedure, which explicitly computes the partial derivatives required for the gradients.

Let's first examine the initial layers in (5) more closely. At Layer-2, we have:

$$\begin{aligned} \boldsymbol{z}^{[2]} &= W^{[2]}\, \boldsymbol{X} + \boldsymbol{b}^{[2]}, \\ \boldsymbol{a}^{[2]} &= \sigma^{[2]}(\boldsymbol{z}^{[2]}) = \sigma^{[2]}\big(W^{[2]}\, \boldsymbol{X} + \boldsymbol{b}^{[2]}\big); \end{aligned}$$

at Layer-3,

$$\begin{aligned} \boldsymbol{z}^{[3]} &= W^{[3]}\, \boldsymbol{a}^{[2]} + \boldsymbol{b}^{[3]} = W^{[3]}\, \sigma^{[2]}\big(W^{[2]}\, \boldsymbol{X} + \boldsymbol{b}^{[2]}\big) + \boldsymbol{b}^{[3]}, \\ \boldsymbol{a}^{[3]} &= \sigma^{[3]}\big(\boldsymbol{z}^{[3]}\big) = \sigma^{[3]}\big(W^{[3]}\, \sigma^{[2]}\big(W^{[2]}\, \boldsymbol{X} + \boldsymbol{b}^{[2]}\big) + \boldsymbol{b}^{[3]}\big); \end{aligned}$$

at Layer-4,

$$\begin{aligned} \boldsymbol{z}^{[4]} &= W^{[4]}\, \boldsymbol{a}^{[3]} + \boldsymbol{b}^{[4]} = W^{[4]}\, \sigma^{[3]}\big(W^{[3]}\, \sigma^{[2]}\big(W^{[2]}\, \boldsymbol{X} + \boldsymbol{b}^{[2]}\big) + \boldsymbol{b}^{[3]}\big) + \boldsymbol{b}^{[4]}, \\ \boldsymbol{a}^{[4]} &= \sigma^{[4]}\big(\boldsymbol{z}^{[4]}\big) = \sigma^{[4]}\big(W^{[4]}\, \sigma^{[3]}\big(W^{[3]}\, \sigma^{[2]}\big(W^{[2]}\, \boldsymbol{X} + \boldsymbol{b}^{[2]}\big) + \boldsymbol{b}^{[3]}\big) + \boldsymbol{b}^{[4]}\big). \end{aligned}$$

Clearly, $\boldsymbol{a}^{[4]}$ involves linear transformations via $W^{[2]}, \boldsymbol{b}^{[2]}$; $W^{[3]}, \boldsymbol{b}^{[3]}$; $W^{[4]}, \boldsymbol{b}^{[4]}$ and the application of the nonlinear transformations $\sigma^{[\ell]}(\cdot)$ for $\ell = 2, 3, 4$, occurring three times. Similarly, at the output Layer-L, $\boldsymbol{a}^{[L]}(\boldsymbol{X}; \boldsymbol{p})$ involves linear transformations via $W^{[2]}, \boldsymbol{b}^{[2]}; \ldots; W^{[L]}, \boldsymbol{b}^{[L]}$, alternating with nonlinear transformations $\sigma^{[\ell]}(\cdot)$ for $\ell = 2, \ldots, L$, for a total of $L - 1$ times.

In general, from (5), we see that at the input Layer-1, $\boldsymbol{a}^{[1]} = \boldsymbol{X}$. For layers $\ell \in \{2, \ldots, L\}$, noting that

$$\boldsymbol{z}^{[\ell]} = W^{[\ell]}\, \boldsymbol{a}^{[\ell-1]} + \boldsymbol{b}^{[\ell]} = W^{[\ell]}\, \sigma^{[\ell-1]}(\boldsymbol{z}^{[\ell-1]}) + \boldsymbol{b}^{[\ell]}, \tag{15}$$

we observe that $\boldsymbol{z}^{[L]}$ is a function of $\boldsymbol{z}^{[L-1]}$, which is a function of $\boldsymbol{z}^{[L-2]}$, and so on. Moreover, we can represent the weight matrix using its column vectors:

$$W^{[\ell]} = (\boldsymbol{w}_{\cdot,1}^{[\ell]}, \ldots, \boldsymbol{w}_{\cdot,n^{[\ell-1]}}^{[\ell]}) \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}},$$

which gives

$$W^{[\ell]}\,\boldsymbol{a}^{[\ell-1]} = \sum_{k=1}^{n^{[\ell-1]}} \boldsymbol{w}_{\cdot,k}^{[\ell]}\, a_k^{[\ell-1]} \in \mathbb{R}^{n^{[\ell]}}. \tag{16}$$

The derivatives of the loss function $C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})$ with respect to $\boldsymbol{p}$ involve the following derivatives:

$$\frac{\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})}{\partial \boldsymbol{a}^{[L]}}, \quad \frac{\partial \boldsymbol{a}^{[\ell]}}{\partial \boldsymbol{z}^{[\ell]}}, \quad \Big(\frac{\partial \boldsymbol{z}^{[\ell]}}{\partial W^{[\ell]}}, \frac{\partial \boldsymbol{z}^{[\ell]}}{\partial \boldsymbol{b}^{[\ell]}}\Big)$$

calculated at L layers. The backpropagation procedure is presented in Proposition 1 below. For illustration, consider the quadratic loss function in (8). The explicit forms of the gradients $\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})/\partial W^{[\ell]}$ in (20) and $\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})/\partial \boldsymbol{b}^{[\ell]}$ in (21), expressed through the explicit forms of

$$\boldsymbol{\delta}^{[\ell]} = \frac{\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})}{\partial \boldsymbol{z}^{[\ell]}} = \Big(\frac{\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})}{\partial z_1^{[\ell]}}, \ldots, \frac{\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})}{\partial z_{n^{[\ell]}}^{[\ell]}}\Big)^{\top} \in \mathbb{R}^{n^{[\ell]}}, \quad \ell = 2, \ldots, L \tag{17}$$

make gradient-based optimization algorithms directly applicable for updating the model parameters.

Proposition 1. *For* $\sigma^{[\ell]}(\boldsymbol{z}^{[\ell]}) = (\sigma^{[\ell]}(z_1^{[\ell]}), \ldots, \sigma^{[\ell]}(z_{n^{[\ell]}}^{[\ell]}))^{\top}$, *define the matrix*

$$D^{[\ell]} = \frac{\partial \sigma^{[\ell]}(\boldsymbol{z}^{[\ell]})}{\partial \boldsymbol{z}^{[\ell]}} = \Big(\frac{\partial \sigma^{[\ell]}(z_1^{[\ell]})}{\partial \boldsymbol{z}^{[\ell]}}, \ldots, \frac{\partial \sigma^{[\ell]}(z_{n^{[\ell]}}^{[\ell]})}{\partial \boldsymbol{z}^{[\ell]}}\Big) \in \mathbb{R}^{n^{[\ell]} \times n^{[\ell]}}, \quad \ell = 2, \ldots, L.$$

Then, for the loss function $C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})$ *in* (8), *we have:*

$$\boldsymbol{\delta}^{[L]} = D^{[L]}\{\boldsymbol{a}^{[L]} - \boldsymbol{Y}\}, \tag{18}$$

$$\boldsymbol{\delta}^{[\ell]} = D^{[\ell]}\,(W^{[\ell+1]})^{\top}\boldsymbol{\delta}^{[\ell+1]}, \quad \ell = L-1, \ldots, 2, \tag{19}$$

and for $\ell = 2, \ldots, L$,

$$\frac{\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})}{\partial W^{[\ell]}} = \boldsymbol{\delta}^{[\ell]} \cdot (\boldsymbol{a}^{[\ell-1]})^{\top}, \tag{20}$$

$$\frac{\partial C_{\boldsymbol{d}}^{[L]}(\boldsymbol{p})}{\partial \boldsymbol{b}^{[\ell]}} = \boldsymbol{\delta}^{[\ell]}, \tag{21}$$

where $\boldsymbol{a}^{[\ell]} = \boldsymbol{a}^{[\ell]}(\boldsymbol{X}; \boldsymbol{p})$ *is defined in the forward-pass* (5).

The utility of the backpropagation algorithm, designed for the quadratic loss function, is demonstrated in Section 3 and Section 4. Proposition 1 can be flexibly modified to accommodate other types of loss functions.

3 Feedforward NN for Binary Classification

In this section, we illustrate the application of feedforward NN to binary classification. Following the discussion in Section 2.1, for a data point $\boldsymbol{d} = (\boldsymbol{X}, \boldsymbol{Y})$, the output in the forward-pass step is $\widehat{F}(\boldsymbol{X}) = (\widehat{F}_1(\boldsymbol{X}), \widehat{F}_2(\boldsymbol{X}))^{\top} = \boldsymbol{a}^{[L]}(\boldsymbol{X}; \widehat{\boldsymbol{p}})$, where the activation function softmax is applied at

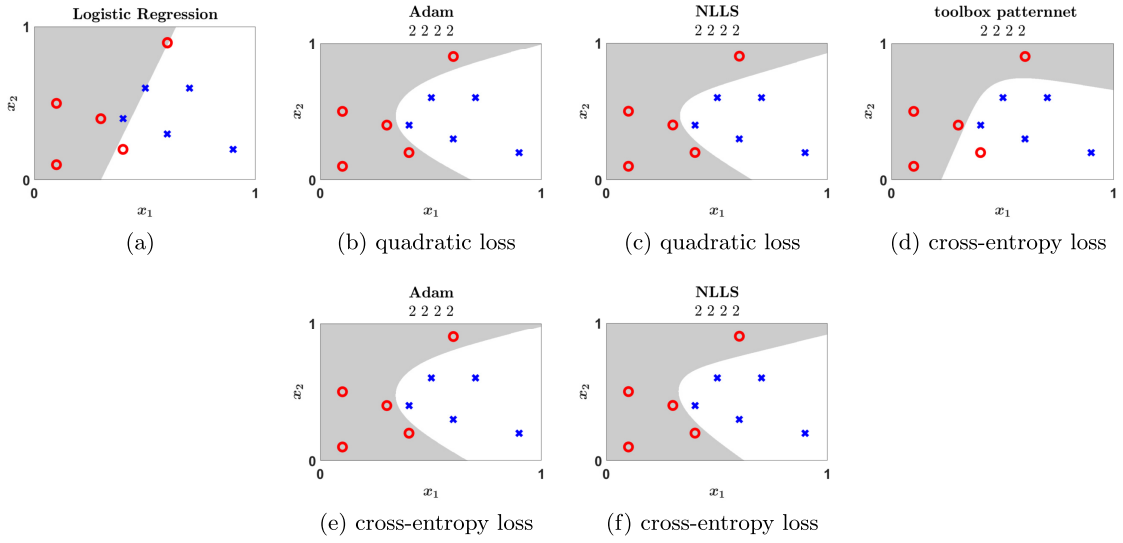
Layer-L, and $\widehat{\boldsymbol{p}}$ represents the parameter set learned from the training dataset. The class label for an input feature $\boldsymbol{X}$ is determined as follows:

$$\widehat{Y}_c = \begin{cases} 1, & \text{if } \widehat{F}_1(\boldsymbol{X}) > \widehat{F}_2(\boldsymbol{X}), \\ 2, & \text{if } \widehat{F}_1(\boldsymbol{X}) < \widehat{F}_2(\boldsymbol{X}). \end{cases}$$

This, in turn, defines the classification boundary curve: $\{\boldsymbol{x} : \widehat{F}_1(\boldsymbol{x}) = \widehat{F}_2(\boldsymbol{x})\}$.

Similar to the toy example in Higham and Higham (2019), Figure 2 plots 10 training points of the input feature $\boldsymbol{X} = (X_1, X_2)^\top$ in a two-dimensional plane, with 5 circles representing class-1 and 5 stars representing class-2. For comparison, the classification boundary obtained by classical logistic regression (McCullagh and Nelder, 1989, using MATLAB's `fitnet` toolbox) is a straight line, which does not effectively separate the two classes. The NN method, configured with 4 layers with node counts $n^{[1]} = 2$, $n^{[2]} = 2$, $n^{[3]} = 2$, and $n^{[4]} = 2$, uses `tanh` activation functions for the hidden layers, quadratic loss and `softmax` for the output layer. Combined with the **Adam** optimizer (see `fmin_adam.m` in Muir (2024)) and backpropagation, this approach produces a nonlinear classification boundary that performs well. In this case, an alternative **NLLS** solver (using MATLAB's `lsqnonlin` function) for the optimization problem (6) also outperforms the logistic regression method. The MATLAB's neural network toolbox `patternnet` (for 'pattern recognition neural network') generates a different classification boundary, using the Scaled Conjugate Gradient (SCG) optimizer with `tansig` activation functions for the hidden layers and cross-entropy loss with `softmax` at the output layer when training a classification model. The bottom panel of Figure 2 adds results using cross-entropy loss for both **Adam** and **NLLS**, which closely resemble those obtained with quadratic loss.

To further examine the performance of the **SGD** and mini-**BGD** algorithms, Figure 3 presents the classification boundaries using different batch sizes m_{B}. The algorithms perform comparably, and the values of the cost functions (with cross-entropy loss) during NN training decrease rapidly to zero as the number of iterations increases.



(a) (b) quadratic loss (c) quadratic loss (d) cross-entropy loss

(e) cross-entropy loss (f) cross-entropy loss

Figure 2: [**Binary classification**] Comparison of classification boundaries, using logistic regression, the **Adam** algorithm, the **NLLS** algorithm, and the MATLAB's `patternnet` toolbox.

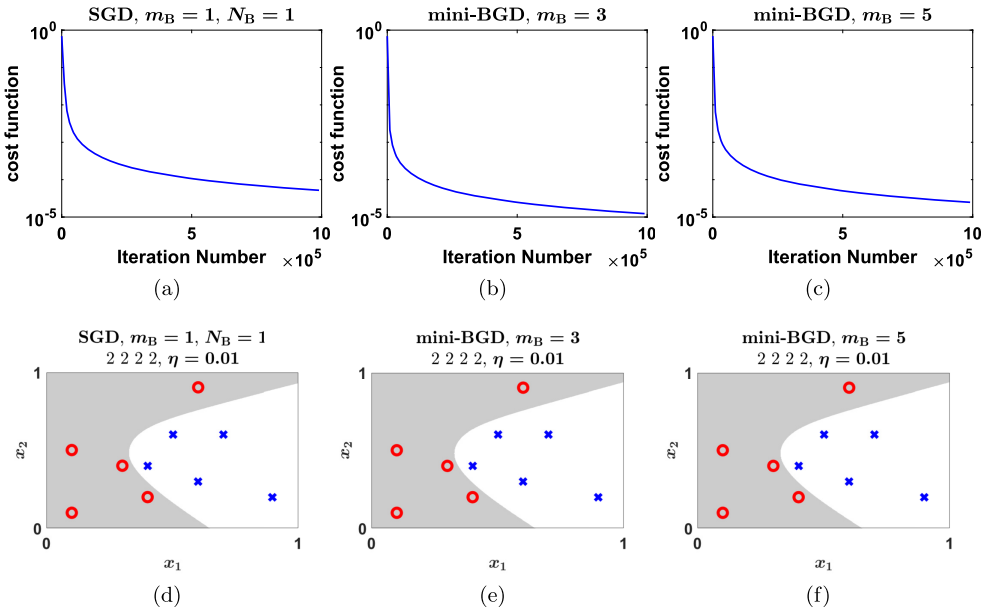


Figure 3: [**Binary classification**] Top: values of the cost functions in (7) versus iterations. Bottom: classification boundaries using **SGD** and various batch sizes m_B in the mini-**BGD** algorithm.

4 Feedforward NN for Nonparametric Function Estimation

This section illustrates the utility of the feedforward NN for estimating the nonparametric regression function in a regression model. For the continuous output variable Y and a univariate input variable X, consider a signal-plus-noise nonparametric regression model:

$$Y = m(X) + \epsilon, \tag{22}$$

where ϵ denotes the noise term and $m(\cdot)$ denotes an unknown regression function. A large number of nonparametric function estimation methods have been developed in the literature, including smoothing splines (Wahba, 1990), regression splines (Friedman, 1991), and the kernel-weighted local polynomial fitting method (Fan, 2018), among others.

In practical applications, the regression curve $m(\cdot)$ can be highly nonlinear, motivating the exploration of NN in nonparametric regression. Let's outline the basic steps. For an individual data point $\boldsymbol{d} = (X, Y)$, the loss function is proportional to the squared error $\{Y - \widehat{m}(X)\}^2$ between the predicted output $\widehat{m}(X)$ and the actual target output Y. As before, the forward-pass step begins with $a^{[1]} = X$. For hidden layers $\ell = 2, \ldots, L-1$, we have $\boldsymbol{z}^{[\ell]} = W^{[\ell]}\boldsymbol{a}^{[\ell-1]} + \boldsymbol{b}^{[\ell]}$, and $\boldsymbol{a}^{[\ell]} = \sigma^{[\ell]}(\boldsymbol{z}^{[\ell]})$. At the output Layer-$L$, we obtain the fitted value $\widehat{m}(X) = z^{[L]}$ by applying a linear activation function $\sigma^{[L]}(z) = z$ to

$$z^{[L]} = W^{[L]}\boldsymbol{a}^{[L-1]} + b^{[L]} = W^{[L]}\sigma^{[L-1]}(\boldsymbol{z}^{[L-1]}) + b^{[L]}.$$

To evaluate the performance of the feedforward NN for curve fitting, we first conduct a simulation study. The training data points $\{(X_i, Y_i) : i = 1, \ldots, 100\}$ are randomly generated from model (22), with the true regression function $m(x) = 5\sin(1/x)$ and a standard Gaussian noise term ϵ, where the input variable X is uniformly distributed on the interval $(0.1, 0.5)$.

The test dataset consists of 100 points of X equally spaced between $\min(X_i)$ and $\max(X_i)$. To implement the NN modeling as outlined in Section 2, Figure 4, panels (a) and (b), adopt the tanh activation function for hidden layers to display the fitted regression functions $\widehat{m}(\cdot)$ at the test input points using the **BGD** and **Adam** algorithms, respectively, along with a scatter plot of the original training points and the true regression function $m(\cdot)$. Our numerical experiments indicate that the number of layers, the number of nodes, the initial values for weights and biases, and the learning rate need to be finely tuned to produce the desired results. For comparison, panel (c) presents the fits using MATLAB's fitnet toolbox for training a 'function fitting neural network', which uses the Levenberg-Marquardt optimizer with the tanh activation function for hidden layers and a linear activation for the output layer, with optimally tuned hyperparameters in advance.

In MATLAB's fitnet function, the gradient descent method is not directly specified. Instead, fitnet uses the Levenberg-Marquardt algorithm by default for training, which is a type of gradient descent method tailored for optimization problems such as training neural networks. For comparison, the number of nodes for all methods is included in the titles to facilitate comparison. Indeed, our implemented **BGD** with a learning rate $\eta = 0.01$ and **Adam** methods compare reasonably well with the toolbox fitnet.

Additionally, the bottom panels of Figure 4 compare the regression functions estimated for real motorcycle data, which contains 133 sample points and is publicly available (see Fan (2018)). For presentational convenience and the choice of learning rates, both the predictor variable and response variable are rescaled to fall within the interval $[0, 1]$. Once again, by appropriately

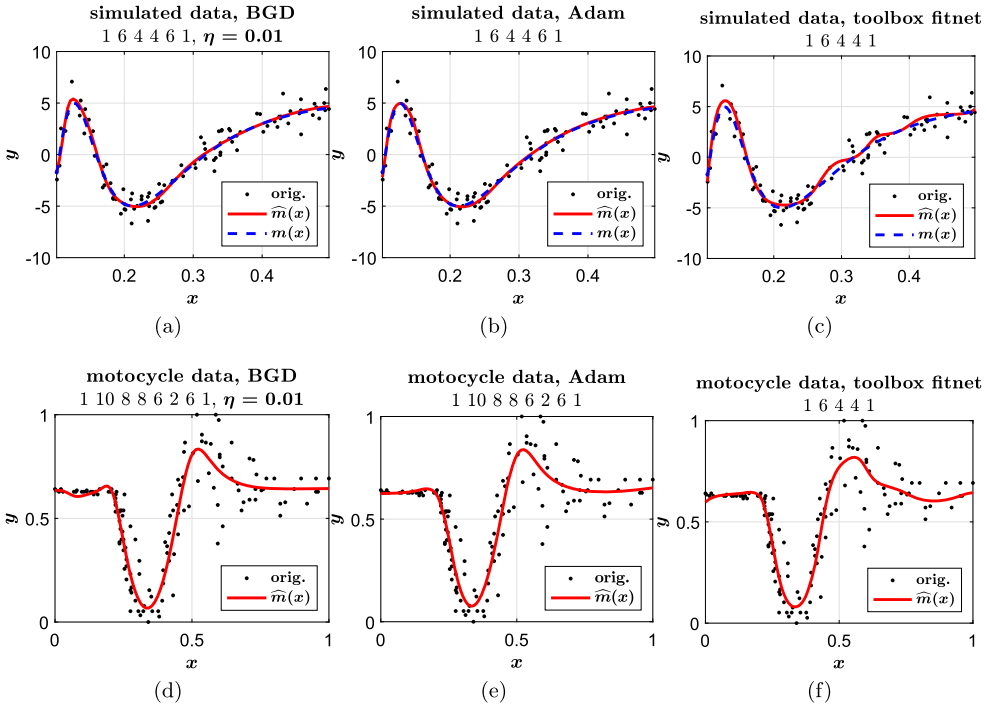


Figure 4: [**Nonparametric regression**] Estimation of the non-linear function $m(x)$. Top row: for the simulated data. Bottom row: for the real motocycle data. Left: using the **BGD** algorithm. Middle: using the **Adam** algorithm. Right: using the MATLAB's fitnet toolbox.

setting the NN architectures, the results using the **BGD** and **Adam** algorithms are comparable to those obtained using the toolbox function fitnet. The fitnet also captures the small oscillation trend of the real data, while the other two methods produce a relatively smooth curve.

Remark 2. As a comparison, traditional approaches like kernel regression typically perform well for low-dimensional input features $\boldsymbol{X}$ and smooth regression functions. Deep learning, on the other hand, excels at handling complex, highly nonlinear structures with high-dimensional variables that may pose challenges for traditional approaches.

5 LSTM for Predicting Sequential Data

Long short-term memory (LSTM) is a type of recurrent neural network (RNN) designed to address the vanishing gradient problem present in traditional RNNs (reviewed in Section 5.1). Its relative insensitivity to gap length gives it an advantage over other RNNs, hidden Markov models, and various sequence learning methods. LSTM enables RNNs to retain information across thousands of timesteps, hence the name 'long short-term memory'. It is applicable to classification, processing, and predicting data based on time series, with applications in handwriting, speech recognition, machine translation, speech activity detection, robotics, video games, and healthcare.

5.1 RNN

In neural networks, a layer performs a transformation from input to output. A single-layer network can be represented as:

$$\boldsymbol{x} \mapsto \quad \boldsymbol{y} = f(W\boldsymbol{x} + \boldsymbol{b}), \tag{23}$$

where f is a linear or nonlinear transfer function (see Figure 5(a)). Single-layer networks cannot effectively handle spatiotemporal sequence data, as they cannot 'remember' previous information in the sequence. Even deep neural networks with multiple hidden layers struggle with this type of data.

To address this limitation, RNNs introduce the concept of a *hidden state*, which extracts features from sequential data and transforms them into outputs. The **hidden state calculation** is given by:

$$(\boldsymbol{x}_1, \boldsymbol{h}_0) \mapsto \quad \boldsymbol{h}_1 = f(U\,\boldsymbol{x}_1 + W\,\boldsymbol{h}_0 + \boldsymbol{b}), \tag{24}$$

$$(\boldsymbol{x}_2, \boldsymbol{h}_1) \mapsto \quad \boldsymbol{h}_2 = f(U\,\boldsymbol{x}_2 + W\,\boldsymbol{h}_1 + \boldsymbol{b}), \tag{25}$$

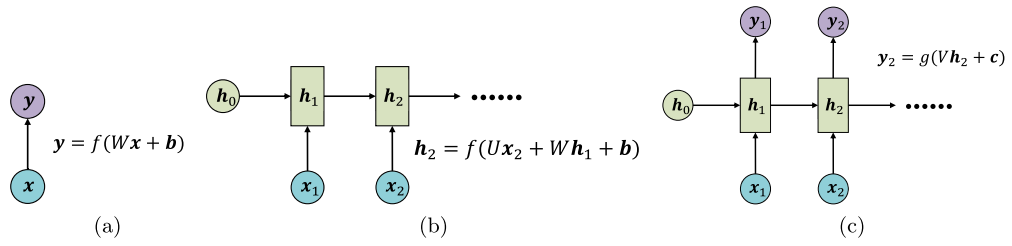


Figure 5: Illustrating steps in the RNN. (a): Single-layer network as shown in (23). (b): Hidden states as in (24) and (25). (c): RNN outputs in (26) and (27).

where $\boldsymbol{h}_0$ is the initial hidden state (often initialized from a standard normal distribution), f is a linear or nonlinear function (typically an activation function such as tanh in (2)), and $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$ represent the input sequence data points (see Figure 5(b)). Unlike deep neural networks, which have many parameters, RNNs share the parameters U, W, and $\boldsymbol{b}$ at each step (as are V and $\boldsymbol{c}$ in the output calculation below).

The RNN **output calculation** is given by:

$$\boldsymbol{h}_1 \mapsto \quad \boldsymbol{y}_1 = g(V\,\boldsymbol{h}_1 + \boldsymbol{c}), \tag{26}$$

$$\boldsymbol{h}_2 \mapsto \quad \boldsymbol{y}_2 = g(V\,\boldsymbol{h}_2 + \boldsymbol{c}), \tag{27}$$

as illustrated in Figure 5(c), where g is a task-specific function such as softmax. The term 'recurrent' in recurrent neural networks (RNNs) arises from the network's iterative processing of sequential data by repeatedly applying the same hidden state module (with shared parameters) at each timestep. In each iteration, the hidden state from the previous step is used along with the current data point as input, enabling hidden states to form a chain and propagate forward. This structure enables RNNs to effectively process sequential data.

However, RNNs have limitations, particularly with long-term dependencies, which refer to situations where predictions require information from much earlier in a sequence. Maintaining long-term memory is critical for such tasks. As each recurrent step in RNNs passes the entire hidden state to the next, weights and information accumulate, leading to the potential forgetting of earlier information and resulting in the vanishing gradient problem during backpropagation. Consequently, RNNs do not inherently provide long-term memory capabilities.

5.2 LSTM

The Long Short-Term Memory (LSTM) module was introduced to address the limitations of RNNs, which lack the ability to retain long-term memory.

In an RNN, the recurrent module is a simple structure, often using the tanh activation function (used as f in (24) and (25)), which constrains values within the range $[-1, 1]$. The LSTM shares a similar framework with RNNs but has a more complex recurrent module structure. While the RNN recurrent module repeats a single layer, the LSTM recurrent module contains three sigmoid layers and one tanh layer, which interact in a unique way (see Figure 6(a)). Here, $\sigma(\cdot)$ denotes the sigmoid activation function. LSTM shares a similar framework with RNNs but differs in the structure of its recurrent module. While the RNN recurrent module repeats a single layer (besides the output layer), the LSTM recurrent module includes three sigmoid layers and one tanh layer, interacting in a unique way. See Figure 6(a) for the LSTM module, where $\sigma(\cdot)$ denotes the sigmoid activation function. The sigmoid activation function, defined in (1), is similar to the tanh function in (2), but its range is [0, 1]. multiplying by 0 discards information, while multiplying by 1 retains it. This allows the LSTM to prioritize important information while discarding less relevant information within its limited memory capacity.

The core of LSTM is the 'cell state' (see Figure 6(b)), which flows along a chain-like path through the LSTM module with minimal linear interactions, maintaining continuity of information. The term 'chain' here is metaphorical, meaning that the same LSTM module is invoked repeatedly at each time step rather than involving multiple distinct LSTM modules.

To manage the cell state effectively, LSTM includes special structures called 'gates' that selectively add or discard information. Each gate consists of a sigmoid layer and a pointwise multiplication operation that filters information flow.

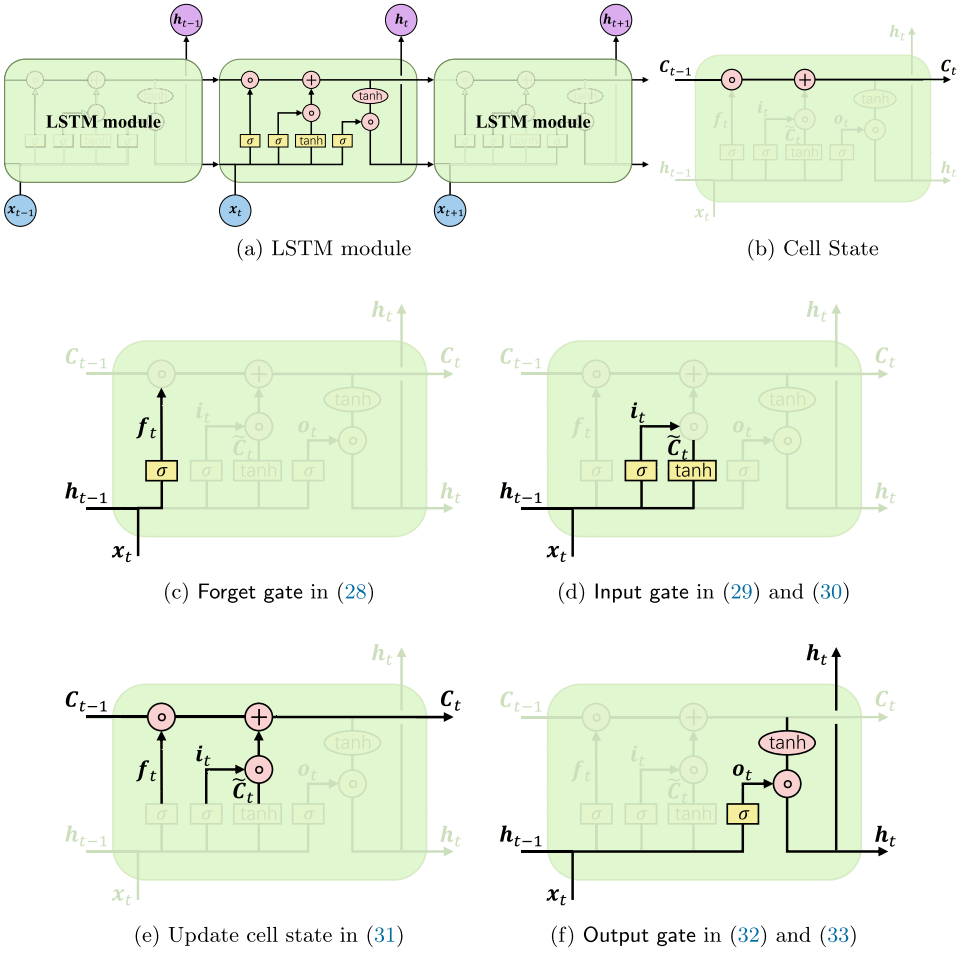


Figure 6: Illustration of steps in LSTM.

The LSTM has three types of gates: the forget gate, the input gate, and the output gate, each contributing to managing the cell state. The **forget gate** is defined as:

$$\boldsymbol{f}_t = \sigma(W_f \cdot (\boldsymbol{h}_{t-1}^\top, \boldsymbol{x}_t^\top)^\top + \boldsymbol{b}_f), \tag{28}$$

where $W_f \cdot (\boldsymbol{h}_{t-1}^\top, \boldsymbol{x}_t^\top)^\top = (W_{fh}, W_{fx}) \cdot (\boldsymbol{h}_{t-1}^\top, \boldsymbol{x}_t^\top)^\top = W_{fh}\,\boldsymbol{h}_{t-1} + W_{fx}\,\boldsymbol{x}_t$ (see Figure 6(c)). The first step in LSTM is to determine what information to discard from the cell state using the forget gate. The forget gate reads the previous output $\boldsymbol{h}_{t-1}$ and the current input $\boldsymbol{x}_t$, applies a sigmoid layer, and outputs a vector $\boldsymbol{f}_t$ with values between 0 and 1. A value of 1 means to fully retain the information, while 0 means to discard it. Finally, this vector $\boldsymbol{f}_t$ is pointwise multiplied with the previous cell state $\boldsymbol{C}_{t-1}$, allowing the model to remember important information while discarding irrelevant data.

The **input gate** is defined as:

$$\boldsymbol{i}_t = \sigma(W_i \cdot (\boldsymbol{h}_{t-1}^\top, \boldsymbol{x}_t^\top)^\top + \boldsymbol{b}_i), \tag{29}$$

$$\widetilde{\boldsymbol{C}}_t = \tanh(W_C \cdot (\boldsymbol{h}_{t-1}^\top, \boldsymbol{x}_t^\top)^\top + \boldsymbol{b}_C). \tag{30}$$

See Figure 6(d). This step determines what new information will be stored in the cell state and consists of two parts: the sigmoid layer, known as the 'input gate', which determines the values

and the extent to which $\widetilde{\boldsymbol{C}}_t$ will be used to update the cell state $\boldsymbol{C}_t$, and the tanh layer, which creates a new candidate value vector $\widetilde{\boldsymbol{C}}_t$ that will be added to $\boldsymbol{C}_t$.

The **update cell state** is given by:

$$\boldsymbol{C}_t = \boldsymbol{f}_t \circ \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \circ \widetilde{\boldsymbol{C}}_t, \tag{31}$$

where '$\circ$' represents the pointwise product, e.g., $\boldsymbol{z}_1 \circ \boldsymbol{z}_2 = (z_{1,1}z_{2,1}, \ldots, z_{1,d}z_{2,d})^\top$ for $\boldsymbol{z}_1 = (z_{1,1}, \ldots, z_{1,d})^\top$ and $\boldsymbol{z}_2 = (z_{2,1}, \ldots, z_{2,d})^\top$ (see Figure 6(e)). In this step, we first perform a pointwise product of the old cell state $\boldsymbol{C}_{t-1}$ with $\boldsymbol{f}_t$, discarding information deemed unnecessary, and then add $\boldsymbol{i}_t \circ \widetilde{\boldsymbol{C}}_t$, the candidate value vector that updates each cell state component as determined by the input gate.

The **output gate** is defined as:

$$\boldsymbol{o}_t = \sigma(W_o \cdot (\boldsymbol{h}_{t-1}^\top, \boldsymbol{x}_t^\top)^\top + \boldsymbol{b}_o), \tag{32}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{C}_t). \tag{33}$$

See Figure 6(f). After passing through the preceding gate structures, we can determine the output value based on the cell state. First, we run a sigmoid layer to obtain $\boldsymbol{o}_t$, which identifies which parts of the cell state should be output. We then apply a tanh layer to the cell state (resulting in values between -1 and 1) and perform a pointwise product with $\boldsymbol{o}_t$ to obtain $\boldsymbol{h}_t$. This ensures that only selected portions of the cell state are output.

Overall, LSTM-RNNs can effectively capture and model long-term dependencies in sequential data. The memory patterns and gating mechanisms in LSTMs allow them to adaptively store and forget information based on the context within the sequence, thereby handling long-term dependencies more effectively.

Appendix B presents numerical experiments for LSTM.

6 CNN for Image Classification

A Convolutional Neural Network (CNN) is a class of deep neural networks widely used for visual imagery analysis. CNNs are particularly effective for tasks such as image recognition, classification, object detection, and even video analysis. Inspired by the structure of the animal visual cortex, CNNs are designed to automatically and adaptively learn spatial hierarchies of features, from low-level details to high-level patterns and categories.

Images possess three key properties that necessitate a specialized architecture. First, images are high-dimensional; for instance, a typical classification image contains 150,528 input dimensions due to its 224×224 RGB values. This results in a vast number of weights—over 22 billion in shallow networks—which increases the challenges in training, memory, and computational resources. Second, pixels in close proximity are statistically correlated, yet fully connected networks and LSTMs do not account for this spatial relationship, treating all input connections equally. Finally, images retain their semantic meaning despite minor geometric shifts, such as moving an image of a tree a few pixels to the left. However, fully connected networks and LSTMs would need to relearn what a tree looks like in each new position. Convolutional layers address these issues by processing local regions independently, using fewer parameters, and sharing them across the image, which makes CNNs far more efficient.

6.1 Convolution Operation

We first provide some background on CNNs.

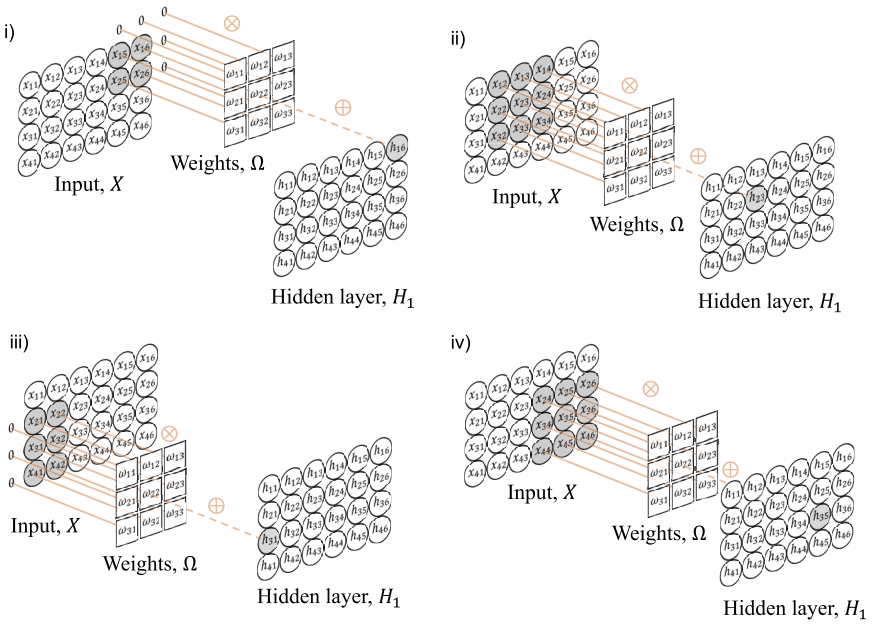


Figure 7: Illustration of CNN. A 2D convolutional layer as described in (34).

6.1.1 Convolutional Layers, Kernel, Stride, and Dilation Rate

For a 2D input image with elements $x_{i,j}$ (for $i = 1, 2, \ldots, p$ and $j = 1, 2, \ldots, q$), a convolutional layer computes its output by convolving the input, adding a bias β, and applying an activation function $\sigma(\cdot)$ to each result. For example, given a $k \times k$ convolution kernel $\Omega = (\omega_{m,n}) \in \mathbb{R}^{k \times k}$, with a stride of 1 and a dilation rate of 0, the convolutional layer computes a single layer of hidden units $h_{i,j}$ (for $i = 1, 2, \ldots, p$ and $j = 1, 2, \ldots, q$) as:

$$h_{i,j} = \sigma(\beta + \textstyle\sum_{m=1}^{k} \sum_{n=1}^{k} \omega_{m,n}\, x_{i+m-k+1, j+n-k+1}). \tag{34}$$

See Figure 7. We typically choose ReLU, as defined in (3), for $\sigma(\cdot)$.

The weights $\omega_{m,n}$ applied at every position (i, j) are collectively called the 'convolution kernel' or 'filter.' The dimensions of the kernel are referred to as the kernel size, which in this example is $k \times k$. The 'stride' is the distance between the centers of two adjacent applications of the kernel. The 'dilation rate' specifies the number of zeros interspersed between the kernel weights, allowing for a broader receptive field. See Figure 8.

6.1.2 Padding

In equation (34), the indices of $x_{i,j}$ can sometimes fall outside the valid range of the input. There are two common approaches to handle this. The first method involves padding the edges of the input with additional values. Zero padding, for example, treats the area outside the input's valid range as zero. Alternatively, the input can be treated as circular or mirrored at the edges. The second method, known as *valid convolution*, discards output areas where the kernel extends beyond the input's range, thereby reducing the size of the output representation without adding extra information at the edges. Examples of zero padding are shown in Figure 7(i) and (iii) and Figure 8.

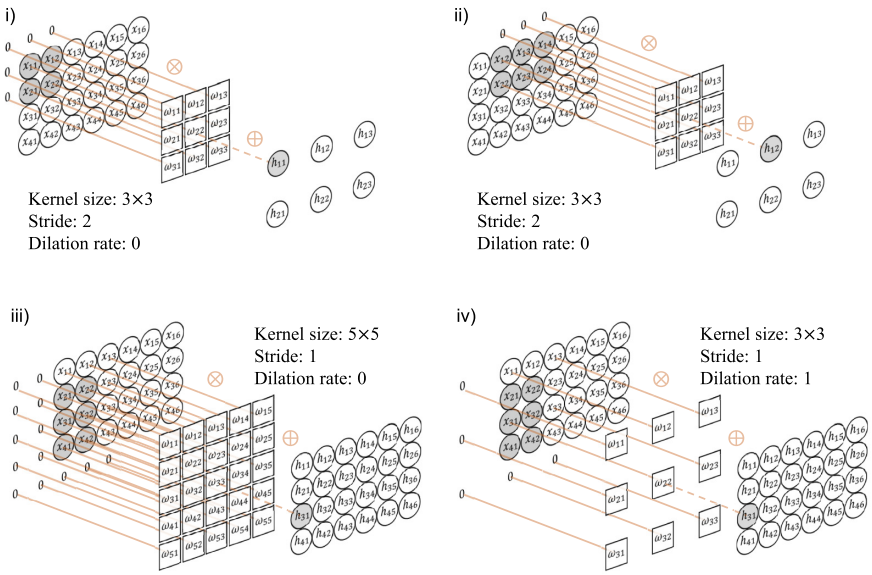


Figure 8: Illustration of CNN. 2D convolutional layers with different kernel sizes, strides, and dilation rates.

6.1.3 Channels

When using a single convolution, some information is naturally lost due to adjacent input averaging and the ReLU activation function, which sets negative values to zero. To address this, multiple convolutions are typically performed simultaneously, with each generating a unique set of hidden variables, referred to as a feature map or 'channel.'

In general, both input and hidden layers contain multiple channels. Kernels across different hidden layers are distinct, and within each kernel, the weights assigned to different input channels are unique. See Figure 9.

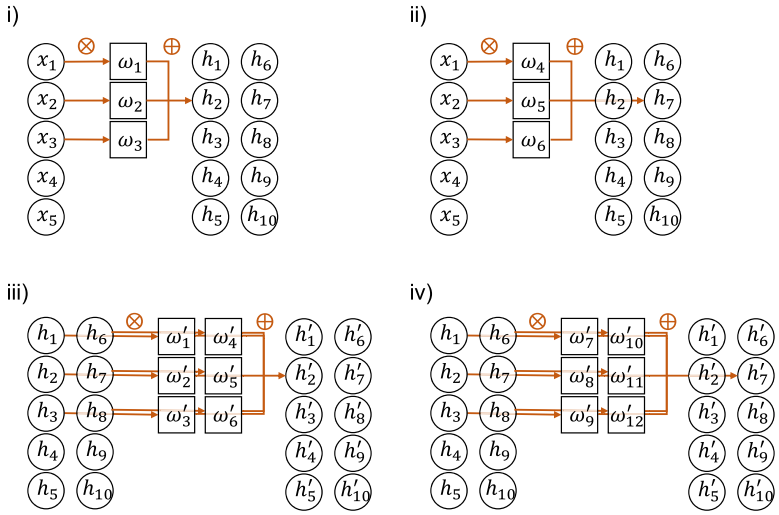


Figure 9: Illustration of CNN. Two 2D convolutional layers viewed from the sides (i) and (ii) are in the same layer, while (iii) and (iv) are in the next layer.

6.2 Downsampling

There are three main approaches to downsampling a 2D representation. Here, we consider the case of scaling down both dimensions by a factor of two. The first method, *strided convolution*, selects every other position in the feature map by applying a convolution with a stride of two (as shown in Figure 10(i)). The second method, *max pooling*, retains the highest value from each 2×2 block of input values (as shown in Figure 10(ii)), offering robustness to shifts in input position, as many maximum values remain the same when the input shifts by a single pixel. The third method, *mean pooling* or *average pooling*, calculates the average value of each 2×2 block of input values (as shown in Figure 10(iii)).

In each of these methods, downsampling is applied independently to each channel, resulting in an output with half the original width and height while preserving the number of channels.

6.3 Numerical Experiments for CNN

The MNIST database of handwritten digits is available at https://yann.lecun.com/exdb/mnist/. This dataset consists of labeled images of handwritten digits, with a training set of 60,000 examples and a test set of 10,000 examples. For our experiments in MATLAB, we use 5,000 samples for training and 5,000 samples for testing, ensuring that each digit appears an equal number of times (500) in both datasets. Figure 11 displays sample images of each digit from the training and test sets. The objective is to classify the digit images.

After training a CNN model on the training dataset using MATLAB's trainNetwork toolbox with the SGDM (**SGD** with momentum) optimizer, we classify the test set images, achieving a prediction accuracy of 99.48% on the test data.

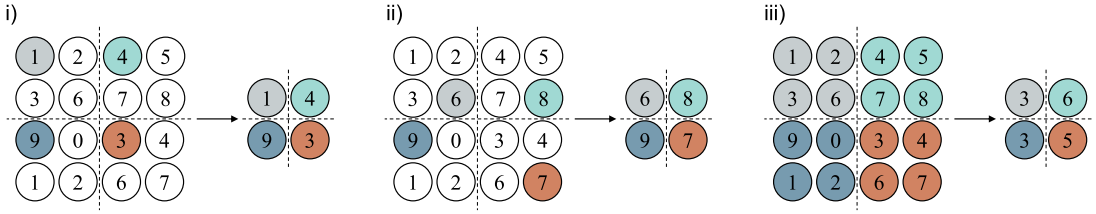
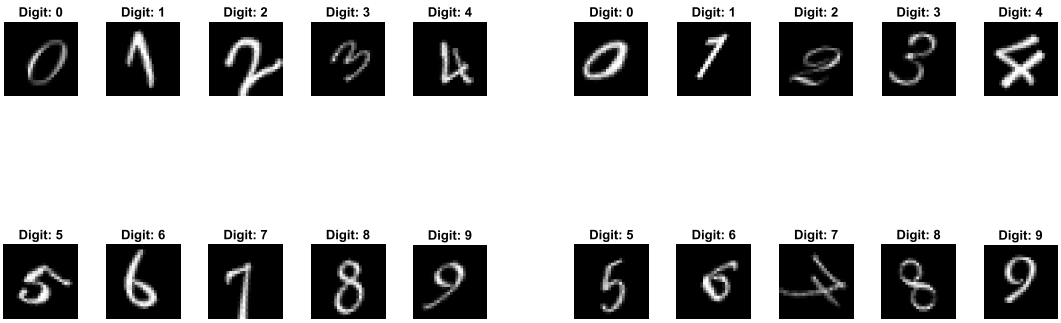


Figure 10: Illustration of three downsampling methods in a CNN.



(a) Training examples (b) Test examples

Figure 11: [MNIST **data**] Panel (a): training examples. Panel (b): test examples.

7 Discussion

The field of deep learning is rapidly evolving, with recent breakthroughs in large language models (LLMs), foundation models, and reinforcement learning. These advancements have revolutionized tasks such as natural language understanding, decision-making, and generalization across domains. While this review includes Feedforward Neural Networks (FNNs), Long Short-Term Memory (LSTM) networks, and Convolutional Neural Networks (CNNs), newer architectures, such as transformers and reinforcement learning agents, have exhibited exceptional performance in applications ranging from text generation to robotics. Furthermore, transfer learning has gained prominence, allowing models to leverage pre-trained knowledge for more efficient learning in specialized tasks.

Despite the remarkable capabilities of deep learning methods, challenges persist, particularly in areas such as model efficiency, interpretability, and data-intensive training requirements. Future directions include making these models more accessible and practical, enhancing their robustness, and integrating them with traditional statistical techniques to improve generalization and interpretability.

Supplementary Material

The MATLAB implementation, including a README file, is available at https://github.com/ChunmingZhangUW/Review-NNM_JDS. The supplementary file includes Appendix A for the proof of Proposition 1 and Appendix B for numerical illustrations of LSTM models in Section 5.2.

Acknowledgement

We thank the Co-Editor and two reviewers for their insightful comments.

Funding

C. Zhang's work was partially supported by the U.S. National Science Foundation grants DMS-2013486 and DMS-1712418, as well as funding provided by the University of Wisconsin-Madison Office of the Vice Chancellor for Research and Graduate Education through the Wisconsin Alumni Research Foundation. Z. Zhang's research was supported by NSFC 72442027.

References

Alzubaidi L, Zhang J, Humaidi AJ, et al. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, 8(1): 53. https://doi.org/10.1186/s40537-021-00444-8

Coleman TF, Li Y (1994). On the convergence of reflective Newton methods for large-scale nonlinear minimization subject to bounds. *Mathematical Programming*, 67(2): 189–224. https://doi.org/10.1007/BF01582221

Fan J (2018). *Local Polynomial Modelling and Its Applications*, Monographs on Statistics and Applied Probability 66. Routledge.

Farrell MH, Liang T, Misra S (2021). Deep neural networks for estimation and inference. *Econometrica*, 89(1): 181–213. https://doi.org/10.3982/ECTA16901

Friedman JH (1991). Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1): 1–67.

Friedman JH, Tibshirani R, Hastie T (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 1st ed. Springer Series in Statistics. Springer, New York.

Goodfellow I, Bengio Y, Courville A (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Higham CF, Higham DJ (2019). Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4): 860–891. https://doi.org/10.1137/18M1165748

Hinton GE (2007). Learning multiple layers of representation. *Trends in Cognitive Sciences*, 11(10): 428–434. https://doi.org/10.1016/j.tics.2007.09.004

Hinton GE, Osindero S, Teh Y-W (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7): 1527–1554. https://doi.org/10.1162/neco.2006.18.7.1527

Jordan MI, Mitchell TM (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245): 255–260. https://doi.org/10.1126/science.aaa8415

Katthi JR, Ganapathy S, Kothinti S, Slaney M (2020). Deep canonical correlation analysis for decoding the auditory brain. In: *2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 3505–3508.

Kingma DP, Ba J (2015). Adam: A method for stochastic optimization. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings* (Y Bengio, Y LeCun, eds.). ArXiv, Ithaca, NY. https://hdl.handle.net/11245/1.505367.

Kramer MA (1991). Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2): 233–243. https://doi.org/10.1002/aic.690370209

Magnus JR, Neudecker H (2019). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. John Wiley & Sons.

McCullagh P, Nelder J (1989). *Generalized Linear Models*, 2nd ed. Chapman and Hall/CRC, Boca Raton, FL.

Muir D (2024). Adam stochastic gradient descent optimization. https://github.com/DylanMuir/fmin_adam.

Ripley BD (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge; New York.

Schmidt-Hieber J (2020). Nonparametric regression using deep neural networks with relu activation function. *The Annals of Statistics*, 48(4): 1875–1897.

Vogl TP, Mangis J, Rigler A, Zink W, Alkon D (1988). Accelerating the convergence of the back-propagation method. *Biological Cybernetics*, 59: 257–263. https://doi.org/10.1007/BF00332914

Wahba G (1990). *Spline Models for Observational Data*. SIAM.

Zhang C, Zhu L, Shen Y (2023). Robust estimation in regression and classification methods for large dimensional data. *Machine Learning*, 112(9): 3361–3411. https://doi.org/10.1007/s10994-023-06349-2

Zhang S, Lu J, Zhao H (2024). Deep network approximation: Beyond relu to diverse activation functions. *Journal of Machine Learning Research*, 25(35): 1–39.

Zhong R, Zhang J, Zhang C (2024). Nonlinear functional principal component analysis using neural networks. arXiv preprint: https://arxiv.org/abs/2306.14388.