Commun. Comput. Phys.
doi: 10.4208/cicp.OA-2019-0168

Vol. **x**, No. x, pp. 1-33
xxx 20xx

Better Approximations of High Dimensional Smooth Functions by Deep Neural Networks with Rectified Power Units

Bo Li[2,1,†], Shanshan Tang[3,†,‡] and Haijun Yu[1,2,*]

[1] *NCMIS & LSEC, Institute of Computational Mathematics and Scientific/Engineering Computing, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China.*
[2] *School of Mathematical Sciences, University of Chinese Academy of Sciences, Beijing 100049, China.*
[3] *China Justice Big Data Institute, Beijing 100043, China.*

Received 30 September 2019; Accepted (in revised version) 24 October 2019

Abstract. Deep neural networks with rectified linear units (ReLU) are getting more and more popular due to their universal representation power and successful applications. Some theoretical progress regarding the approximation power of deep ReLU network for functions in Sobolev space and Korobov space have recently been made by [D. Yarotsky, Neural Network, 94:103-114, 2017] and [H. Montanelli and Q. Du, SIAM J Math. Data Sci., 1:78-92, 2019], etc. In this paper, we show that deep networks with rectified power units (RePU) can give better approximations for smooth functions than deep ReLU networks. Our analysis bases on classical polynomial approximation theory and some efficient algorithms proposed in this paper to convert polynomials into deep RePU networks of optimal size with no approximation error. Comparing to the results on ReLU networks, the sizes of RePU networks required to approximate functions in Sobolev space and Korobov space with an error tolerance ε, by our constructive proofs, are in general $\mathcal{O}(\log\frac{1}{\varepsilon})$ times smaller than the sizes of corresponding ReLU networks constructed in most of the existing literature. Comparing to the classical results of Mhaskar [Mhaskar, Adv. Comput. Math. 1:61-80, 1993], our constructions use less number of activation functions and numerically more stable, they can be served as good initials of deep RePU networks and further trained to break the limit of linear approximation theory. The functions represented by RePU networks are smooth functions, so they naturally fit in the places where derivatives are involved in the loss function.

[†] The first two authors contributed equally. Author list is alphabetical.

[‡] The work of this author is partially done during her Ph.D. study in Academy of Mathematics and Systems Science, Chinese Academy of Sciences.

[*] Corresponding author. *Email addresses:* libo1171309228@lsec.cc.ac.cn (B. Li), tangshanshan@lsec.cc.ac.cn (S. Tang), hyu@lsec.cc.ac.cn (H. Yu)

AMS subject classifications: 65D15, 65M12, 65M15

Key words: Deep neural network, high dimensional approximation, sparse grids, rectified linear unit, rectified power unit, rectified quadratic unit.

1 Introduction

Artificial neural network(ANN), whose origin may date back to the 1940s [1], is one of the most powerful tools in the field of machine learning. Especially, it became dominant in a lot of applications after the seminal works by Hinton et al. [2] and Bengio et al. [3] on efficient training of deep neural networks (DNNs), which pack up multi-layers of units with some nonlinear activation function. Since then, DNNs have greatly boosted the developments in different areas including image classification, speech recognition, computational chemistry and numerical solutions of high-dimensional partial differential equations and scientific problems, etc., see e.g. [4–12] to name a few.

The success of DNNs relies on two facts: 1) DNN is a powerful tool for general function approximation; 2) Efficient training methods are available to find minimizers with good generalization ability. In this paper, we focus on the first fact. It is known that artificial neural networks can approximate any C^0 and L^1 functions with any given error tolerance, using only one hidden layer (see e.g. [13,14]). However, it was realized recently that deep networks have better representation power(see e.g. [15–17]) than shallow networks. One of the commonly used activation functions with DNN is the so called rectified linear unit (ReLU) [18], which is defined as $\sigma(x) = \max(0,x)$. Telgarsky [16] gave a simple and elegant construction showing that for any k, there exist k-layer, $\mathcal{O}(1)$ wide ReLU networks on one-dimensional data, which can express a sawtooth function on $[0,1]$ with $\mathcal{O}(2^k)$ oscillations. Moreover, such a rapidly oscillating function cannot be approximated by poly(k)-wide ReLU networks with $o(k/\log(k))$ depth. Following this approach, several other works proved that deep ReLU networks have better approximation power than shallow ReLU networks [19–22]. In particular, for C^β-differentiable d-dimensional functions, Yarotsky [21] proved that the number of parameters needed to achieve an error tolerance of ε is $\mathcal{O}(\varepsilon^{-\frac{d}{\beta}}\log\frac{1}{\varepsilon})$. Petersen and Voigtlaender [22] proved that for a class of d-dimensional piecewise C^β continuous functions with the discontinuous interfaces being C^β continuous also, one can construct a ReLU neural network with $\mathcal{O}((1+\frac{\beta}{d})\log_2(2+\beta))$ layers, $\mathcal{O}(\varepsilon^{-\frac{2(d-1)}{\beta}})$ nonzero weights to achieve ε-approximation. The complexity bound is sharp. For analytic functions, E and Wang [23] proved that using ReLU networks with fixed width $d+4$, to achieve an error tolerance of ε, the depth of the network depends on $\log\frac{1}{\varepsilon}$ instead of ε itself. We also want to mention that the detailed relations between ReLU networks and linear finite elements have been studied by He et al. [24]. And recent work by Opschoor, Peterson and Schwab [25] reveals the connection between ReLU DNNs and high-order finite element methods.

One basic fact on deep ReLU networks is that function x^2 can be approximated within any error $\varepsilon > 0$ by a ReLU network having the depth, the number of weights and computation units all of order $\mathcal{O}(\log\frac{1}{\varepsilon})$. This fact has been used by several groups (see e.g. [19, 21]) to analyze the approximation property of general smooth functions using ReLU networks. In this paper, we extend the analysis to deep neural networks using rectified power units (RePUs), which are defined as

$$\sigma_s(x) = \begin{cases} x^s, & x \geq 0, \\ 0, & x < 0, \end{cases} \quad s \in \mathbb{N}_0, \tag{1.1}$$

where $\mathbb{N}_0$ denotes the set of non-negative integers. Note that σ_1 is the commonly used ReLU function, σ_0 is the binary step function. We call σ_2, σ_3 rectified quadratic unit (ReQU) and rectified cubic unit (ReCU), respectively. We show that deep neural networks using RePUs($s \geq 2$) as activation functions have better approximation property for smooth functions than those using ReLUs. By replacing ReLU with RePU($s \geq 2$), the functions x, x^2 and xy can be exactly represented with no approximation error using networks having just a few nodes and nonzero weights. Based on this, we build efficient algorithms to explicitly convert functions from a polynomial space into RePU networks having approximately the same number of coefficients. This allows us to obtain a better upper bound of the best neural network approximation for general smooth functions using classical polynomial approximation theories. Note that σ_s networks have been used in the classic works by Mhaskar and his coworkers (see e.g. [26–28]), where by converting spline approximations into σ_s DNNs, quasi-optimal theoretical upper bounds of function approximation are obtained. However, their constructions of neural network are not optimal for very smooth functions (the case $k \gg s$), the error bound obtained is quasi-optimal due to an extra $\log_s(k)$ factor, where k is related to the smoothness of the underlying functions. Meanwhile no numerically efficient and stable algorithm is presented. In this paper, we present numerically stable and efficient constructions of RePU network representation of polynomials which result in RePU network of different structure and remove the extra $\log_s(k)$ factor in the approximation bounds. After this paper is put on arXiv, the RePU networks and our optimal network constructions are adopted by other authors, e.g. by using deep RePU networks instead of ReLU networks, a sharper bound for approximating holomorphic maps in high dimension is obtained by Opschoor, Schwab and Zech [29].

For high dimensional problems, to be tractable, the intrinsic dimension usually do not grow as fast as the observation dimension. In other words, the problems have low dimensional structure. A particular example is the class of high-dimensional smooth functions with bounded mixed derivatives, for which sparse grid (or hyperbolic cross) approximation is a very popular approximation tool [30–34]. In the past few decades, sparse grid method and hyperbolic cross approximations have found many applications, such as numerical integration and interpolation [30, 35–37], solving partial differential equations (PDE) [38–43], computational chemistry [32, 44–46], uncertainty quantification [47–49], etc. For high dimensional problem, we will derive upper bounds of RePU DNN approximation error by converting sparse grid and hyperbolic cross spectral approximation into RePU networks. Our work is inspired by the recent work of Montanelli and Du [50], where the connection between linear finite element sparse grids and deep ReLU neural networks is established. In this paper, we approximate multivariate functions in high order Korobov space using sparse grid Chebyshev interpolation [36] for the interpolation problem, and using hyperbolic cross spectral approximation for the projection problem [33, 40]. Then, we convert the high-dimensional polynomial approximations into ReQU networks, instead of ReLU networks, to avoid adding an extra factor $\log\frac{1}{\varepsilon}$ in the size of the neural network.

In summary, we find that RePU networks have the following good properties:

- RePU neural networks provide better approximations for sufficient smooth functions comparing to ReLU neural network approximations. To achieve same accuracy, the RePU network approximation we constructed needs less number of layers and smaller network size than existing ReLU neural network approximations. For example, for a function with all the partial derivatives bounded uniformly independent of derivative order, we can construct a ReQU network with no more than $\mathcal{O}\left(\log_2\left(\log\frac{1}{\varepsilon}\right)\right)$ layers, and no more than $\mathcal{O}\left(\frac{\log(1/\varepsilon)}{\log(\log 1/\varepsilon)}\right)$ nonzero weights to approximate it with error ε. More results are given in Theorems 2.4, 3.3, 4.2.

- The functions represented by RePU($s\geq 2$) networks are smooth functions, so they naturally fit in the places where derivatives are involved in the loss function.

- Compared to other high-order differentiable activation functions, such as logistic, tanh, softplus, sinc etc., RePUs are more efficient in terms of number of arithmetic operations needed to evaluate, especially the rectified quadratic unit.

Based on the facts above, we advocate the use of deep RePU networks in places where the functions to be approximated are smooth.

The remaining part of this paper is organized as follows. In Section 2, we first show how to approximate univariate smooth functions using RePU networks by converting best polynomial approximations into RePU networks. Then we use a similar approach to analyze the ReQU network approximation for multivariate functions in weighted Sobolev space in Section 3. After that, we show how high-dimensional functions with sparse polynomial approximations can be well approximated by ReQU networks in Section 4. Some preliminary numerical results are given in Section 5. We end the paper by a short summary in Section 6.

2 Approximation of univariate functions by deep RePU networks

We first introduce some notations related to neural networks. Denote by $\mathbb{N}$ the set of all positive integers, $\mathbb{N}_0:=\{0\}\cup\mathbb{N}$. Given $d,L\in\mathbb{N}$, we denote a neural network Φ with input of dimension d, number of layer L, by a matrix-vector sequence

$$\Phi=\big((A_1,b_1),\cdots,(A_L,b_L)\big), \tag{2.1}$$

where $N_0=d$, $N_1,\cdots,N_L\in\mathbb{N}$, A_k are $N_k\times N_{k-1}$ matrices, and $b_k\in\mathbb{R}^{N_k}$. If Φ is a neural network, and $\rho:\mathbb{R}\to\mathbb{R}$ is an arbitrary activation function, then define

$$R_\rho(\Phi):\mathbb{R}^d\to\mathbb{R}^{N_L}, \qquad R_\rho(\Phi)(\boldsymbol{x})=\boldsymbol{x}_L, \tag{2.2}$$

where $R_\rho(\Phi)(\boldsymbol{x})$ is given as

$$\begin{cases}\boldsymbol{x}_0:=\boldsymbol{x},\\ \boldsymbol{x}_k:=\rho(A_k\boldsymbol{x}_{k-1}+b_k), \quad k=1,2,\cdots,L-1,\\ \boldsymbol{x}_L:=A_L\boldsymbol{x}_{L-1}+b_L,\end{cases} \tag{2.3}$$

and

$$\rho(\boldsymbol{y}):=\Big(\rho(y^1),\cdots,\rho(y^m)\Big), \quad \forall\, \boldsymbol{y}=(y^1,\cdots,y^m)\in\mathbb{R}^m.$$

We use three quantities to measure the complexity of the neural network: number of hidden layers, number of nodes (i.e. activation units), and number of nonzero weights, which are $L-1$, $\sum_{k=1}^{L-1}N_k$ and number of non-zeros in $\{(A_k,b_k),k=1,\cdots,L\}$, respectively, for the neural network defined in (2.1). For convenience, we denote by $\#A$ the number of nonzero components in A for a given matrix or vector A. For the neural network Φ defined in (2.1), we also denote its number of nonzero weights as $\#\Phi:=\sum_{k=1}^{L}(\#A_k+\#b_k)$.

In this paper we study the approximation property of smooth functions by deep neural networks with RePUs as activation units. It seems that σ_s networks were first used in the classic works by Mhaskar and his coworkers (see e.g. [26,27]) to obtain high-order convergence of neural network approximation. σ_s is also a special case of piece-wise polynomial activation function, which has been studied in [51] for shallow network approximation. We also note that σ_3 has been used in a deep Ritz method proposed recently to solve PDEs using variational form [52].

The construction of RePU networks adopted by Mhaskar bases on the fact that a polynomial of degree n in d dimension can be represented by a linear combination of $\binom{n+d}{d}$ number of monomials of the form $\big(Ax+b\big)^n$, with each one using different affine transform. To represent a polynomial of degree n using σ_s neural network, they first compose $\sigma_s(x)$ for $k=\lceil\log_s n\rceil$ times, which result in $\sigma_{s^k}(x)$. Then a neural network with one-layer $\sigma_{s^k}(x)$ units of amount $\binom{n+d}{d}$ is capable to accurately represent any polynomial of degree n. This kind of construction give an optimal linear approximation result for neural network using high order (the order is s^k) sigmoidal activation functions. However, if regard the constructed neural network as a σ_s neural network, it has k hidden layers. The corresponding linear approximation bound is quasi-optimal due to this factor k. Moreover, to find the corresponding network coefficients to represent a given polynomial, one needs to solve a Vandermonde-like matrix, whose condition number is known grows geometrically (see e.g. [53]). In this paper, we propose a different approach which does not involve any Vandermonde matrix of large size.

2.1 Approximation by deep ReQU networks

Our analyses relies upon the fact: x, x^2, $\cdots$, x^s, and xy all can be realized by σ_s neural networks with a few number of coefficients. We first give the result for $s=2$ case.

Lemma 2.1. *For any $x,y\in\mathbb{R}$, the following identities hold:*

$$x^2=\beta_2^T\sigma_2(\omega_2 x), \tag{2.4}$$

$$x=\beta_1^T\sigma_2(\omega_1 x+\gamma_1), \tag{2.5}$$

$$xy=\beta_1^T\sigma_2(\omega_1 x+\gamma_1 y), \tag{2.6}$$

where

$$\beta_2=[1,1]^T,\ \omega_2=[1,-1]^T,\ \beta_1=\frac{1}{4}[1,1,-1,-1]^T,\ \omega_1=[1,-1,1,-1]^T,\ \gamma_1=[1,-1,-1,1]^T. \tag{2.7}$$

If both x and y are non-negative, the formula for x^2 and xy can be simplified to the following form

$$x^2=\sigma_2(x), \tag{2.8}$$

$$xy=\beta_3^T\sigma_2(\omega_3 x+\gamma_2 y), \tag{2.9}$$

where

$$\beta_3=\frac{1}{4}[1,-1,-1]^T,\quad \omega_3=[1,1,-1]^T,\quad \gamma_2=[1,-1,1]^T. \tag{2.10}$$

Proof. All the identities can be obtained by straightforward calculations. □

Note that the realizations given in Lemma 2.1 are not unique. For example, to realize $id_{\mathbb{R}}(x)=x$, we may use

$$x=(x+1/2)^2-x^2-1/4=\beta_2^T\sigma_2(\omega_2(x+1/2))-\beta_2^T\sigma_2(\omega_2 x)-1/4,$$

for general $x\in\mathbb{R}$, and use

$$x=(x+1/2)^2-x^2-1/4=\sigma_2(x+1/2)-\sigma_2(x)-1/4,$$

for non-negative x. To have a neat presentation, we will use (2.4)-(2.10) throughout this paper even though simpler realizations may exist for some special cases. We notice that the realization of the identity map $id_{\mathbb{R}}(x)$ given in (2.5) is a special case of (2.6) with $y=1$. Furthermore, the constant function 1 can be represented by a trivial network with $L=1$ and $A_1=0$, $b_1=1$.

Remark 2.1. Notice that in [21, 22, 50], all the analyses rely on the fact that x^2 can be approximated to an error tolerance ε by a deep ReLU networks of complexity $\mathcal{O}(\log\frac{1}{\varepsilon})$. In our approach, by replacing ReLU with ReQU, x^2 is represented with no error using a ReQU network with only one hidden layer and 2 hidden neurons. This simple replacement greatly simplifies the proofs of some existing deep neural network approximation bounds, improves the approximation rate and meanwhile reduces the network complexity.

2.1.1 Optimal realizations of polynomials by deep ReQU networks with no error

The basic property of σ_2 given in Lemma 2.1 can be used to construct deep neural network representations of monomials and polynomials. We first show that the monomial $x^n, n > 2$ can be represented exactly by deep ReQU networks of finite size but not shallow ReQU networks.

Theorem 2.1. A) *The monomial $x^n, n \in \mathbb{N}$ defined on $\mathbb{R}$ can be represented exactly by a σ_2 network. The number of network layers, number of hidden nodes and number of nonzero weights required to realize x^n are at most $\lfloor \log_2 n \rfloor + 2$, $5\lfloor \log_2 n \rfloor + 5$ and $25\lfloor \log_2 n \rfloor + 14$, respectively. Here $\lfloor x \rfloor$ represents the largest integer not exceeding x for $x \in \mathbb{R}$.*

B) *For any $n > 2$, x^n can not be represented exactly by any ReQU network with less than $\lceil \log_2 n \rceil$ hidden layers.*

Proof. 1) We first prove part B. For a one-layer ReQU network with N activation units, one input and one output, the function represented by the network can be written as

$$f_N(x) = \sum_{k=1}^{N} c_k \sigma_2(a_k x + b_k) + d,$$

where d and a_k, b_k, c_k, $k = 1, \cdots, N$ are the parameters of the network. Obviously, f_N is a piecewise polynomial with at most $N+1$ pieces in the intervals divided by distinct points of $x_k = -b_k/a_k$, $k = 1, \cdots, N$(suppose the points are in ascending order). In each piece, f_N is a polynomials of degree 2. Since a polynomial of degree at most 2 composed with another polynomial of degree at most 2 produces a polynomial of degree at most 4, so a ReQU network with two hidden layers can only represent piecewise polynomials of degree at most 4. By induction, a ReQU network with m hidden layers can only represent piecewise polynomials of degree at most 2^m. So, with $m < \lceil \log_2 n \rceil$, a ReQU network with m hidden layers can't exactly represent x^n.

2) Now we give a constructive proof for part A. We first express n in binary system as follows:

$$n = a_m \cdot 2^m + a_{m-1} \cdot 2^{m-1} + \cdots + a_1 \cdot 2 + a_0,$$

where $a_j \in \{0,1\}$ for $j = 0, 1, \cdots, m-1$, $a_m = 1$, and $m = \lfloor \log_2 n \rfloor$. Then

$$x^n = x^{2^m} \cdot x^{\sum_{j=0}^{m-1} a_j 2^j}.$$

Introducing intermediate variables

$$\xi_k^{(1)} := x^{2^k}, \quad \xi_k^{(2)} := x^{\sum_{j=0}^{k-1} a_j 2^j}, \quad \text{for } 1 \le k \le m,$$

then

$$x^n = \xi_m^{(1)} \xi_m^{(2)}. \tag{2.11}$$

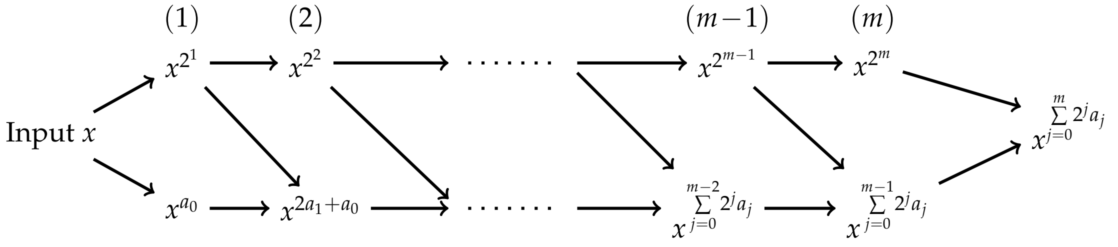

Figure 1: A schematic diagram for σ_2 network realization of x^n. (j) represents the j-th hidden layer for intermediate variables.

We use the iteration scheme

$$\begin{cases} \xi_1^{(1)} = x^2, \\ \xi_1^{(2)} = x^{a_0}, \end{cases} \quad \text{and} \quad \begin{cases} \xi_k^{(1)} = (\xi_{k-1}^{(1)})^2, \\ \xi_k^{(2)} = (\xi_{k-1}^{(1)})^{a_{k-1}} \xi_{k-1}^{(2)}, \end{cases} \quad \text{for } 2 \le k \le m, \tag{2.12}$$

and (2.11) to realize x^n. The outline of the realization is demonstrated in Fig. 1. In each iteration step, we need to realize two basic operations: $(x)^2$ and $(x)^{a_k} y$, where x,y stands for $\xi_{k-1}^{(1)}, \xi_{k-1}^{(2)}$ respectively. Note that $(x)^2$ can be realized by Eqs. (2.4) and (2.8) in Lemma 2.1. For the operation $(x)^{a_j} y$, since $a_j \in \{0,1\}$, by (2.6), we have

$$x^{a_j} y = \Big(\frac{1+(-1)^{a_j}}{2} + \frac{1-(-1)^{a_j}}{2} x \Big) y = \beta_1^T \sigma_2 \Big(\omega_1 (c_j^+ + c_j^- x) + \gamma_1 y \Big), \tag{2.13}$$

where $c_j^{\pm} := \frac{1 \pm (-1)^{a_j}}{2}$.

Now we describe the procedure in detail. For $n \ge 3$, we follow the idea given in Eq. (2.12) and Fig. 1. The function x^n is realized in $m+1$ steps, which are discussed below.

1) In Step 1, we calculate

$$\xi_1^{(1)} = x^2 = \beta_2^T \sigma_2(\omega_2 x) \ge 0, \tag{2.14}$$

$$\xi_1^{(2)} = x^{a_0} = c_0^+ + c_0^- x = c_0^+ + c_0^- \beta_1^T \sigma_2(\omega_1 x + \gamma_1), \tag{2.15}$$

which implies the first layer output of the neural network is:

$$\boldsymbol{x}_1 = \sigma_2(A_1 x + b_1), \quad \text{where} \quad A_1 = \begin{bmatrix} \omega_2 \\ \omega_1 \end{bmatrix}_{6\times 1}, \quad b_1 = \begin{bmatrix} \boldsymbol{0} \\ \gamma_1 \end{bmatrix}_{6\times 1}, \tag{2.16}$$

and

$$\begin{bmatrix} \xi_1^{(1)} \\ \xi_1^{(2)} \end{bmatrix} = A_{20} \boldsymbol{x}_1 + b_{20}, \quad \text{where} \quad A_{20} = \begin{bmatrix} \beta_2^T & \boldsymbol{0} \\ \boldsymbol{0} & c_0^- \beta_1^T \end{bmatrix}_{2\times 6}, \quad b_{20} = \begin{bmatrix} 0 \\ c_0^+ \end{bmatrix}_{2\times 1}. \tag{2.17}$$

Since $\#\omega_1 = 4$, $\#\omega_2 = 2$, $\#\gamma_1 = 4$, it is easy to see that the number of nodes in the first hidden layer is 6, and the number of non-zeros is: $\#A_1 + \#b_1 = 10$.

2) In Step j, $2\le j\le m$, we calculate

$$\begin{aligned}\xi_j^{(1)} &= (\xi_{j-1}^{(1)})^2\\ &= \sigma_2(\xi_{j-1}^{(1)}) \ge 0, \end{aligned} \tag{2.18}$$

$$\begin{aligned}\xi_j^{(2)} &= (\xi_{j-1}^{(1)})^{a_{j-1}}\xi_{j-1}^{(2)}\\ &= (c_{j-1}^{+}+c_{j-1}^{-}\xi_{j-1}^{(1)})\xi_{j-1}^{(2)}\\ &= \beta_1^T\sigma_2\left(\omega_1(c_{j-1}^{+}+c_{j-1}^{-}\xi_{j-1}^{(1)})+\gamma_1\xi_{j-1}^{(2)}\right), \end{aligned} \tag{2.19}$$

which suggest the j-th layer output of the neural network is:

$$\boldsymbol{x}_j=\sigma_2\left(A_{j1}\begin{bmatrix}\xi_{j-1}^{(1)}\\ \xi_{j-1}^{(2)}\end{bmatrix}+b_{j1}\right),\quad A_{j1}=\begin{bmatrix}1 & 0\\ c_{j-1}^{-}\omega_1 & \gamma_1\end{bmatrix}_{5\times 2},\quad b_{j1}=\begin{bmatrix}0\\ c_{j-1}^{+}\omega_1\end{bmatrix}_{5\times 1},$$

and

$$\begin{bmatrix}\xi_j^{(1)}\\ \xi_j^{(2)}\end{bmatrix}=A_{j+1,0}\boldsymbol{x}_j+b_{j+1,0},\quad \text{where}\quad A_{j+1,0}=\begin{bmatrix}1 & \boldsymbol{0}\\ 0 & \beta_1^T\end{bmatrix}_{2\times 5},\quad b_{j+1,0}=\boldsymbol{0}. \tag{2.20}$$

We have

$$A_j=A_{j1}A_{j0},\quad b_j=A_{j1}b_{j0}+b_{j1},\quad j=2,\cdots,m. \tag{2.21}$$

By a direct calculation, we find that the number of nodes in Layer j is 5 $(j=2,\cdots,m)$, and the number of non-zeros in Layer j, $j=3,\cdots,m$ is $\#A_j+\#b_j\le 21+4=25$. For $j=2$, $\#A_2+\#b_2=26+4=30$.

3) In Step $m+1$, we calculate

$$x^n=\xi_m^{(1)}\xi_m^{(2)}=\beta_1^T\sigma_2\left(\omega_1\xi_m^{(1)}+\gamma_1\xi_m^{(2)}\right), \tag{2.22}$$

which implies

$$\boldsymbol{x}_{m+1}=\sigma_2\left(A_{m+1,1}\begin{bmatrix}\xi_m^{(1)}\\ \xi_m^{(2)}\end{bmatrix}\right),\quad \text{where}\quad A_{m+1,1}=[\omega_1\ \gamma_1]_{4\times 2}. \tag{2.23}$$

So we get $\boldsymbol{x}_{m+1}=\sigma_2(A_{m+1}\boldsymbol{x}_m+b_{m+1})$, with

$$A_{m+1}=A_{m+1,1}A_{m+1,0},\quad b_{m+1}=\boldsymbol{0}, \tag{2.24}$$

and

$$\boldsymbol{x}_{m+2}:=x^n=\beta_1^T\boldsymbol{x}_{m+1}. \tag{2.25}$$

By a direct calculation, we get the number of nodes in Layer $m+1$ is 4, the number of non-zero weights is $\#A_{m+1}=20$.

For Layer $m+2$, which is the output layer of the overall network, $A_{m+2}=\beta_1^T$, and $b_{m+2}=0$. There are no activation units and the number of nonzero weights is $\#A_{m+2}=4$.

The ReQU network we just built has $m+2$ layers. The total number of nodes is $6+5(m-1)+4=5m+5$. The total number of nonzero weights is $10+30+25(m-2)+20+4=25m+14$. Combining the cases $n=1,2$, we reach to the desired conclusion. □

Now we consider how to convert univariate polynomials into σ_2 networks. If we directly apply Theorem 2.1 to each monomial term in a polynomial and then combine them together, one would obtain a network of depth $\mathcal{O}(\log_2 n)$ and size $\mathcal{O}(n\log_2 n)$, which is not optimal. We provide here two algorithms to convert a polynomial into a ReQU network of same scale, i.e. without the extra $\log_2 n$ factor. The first algorithm is a direct implementation of Horner's method (also known as Qin Jiushao's algorithm in China):

$$\begin{aligned} f(x)&=a_0+a_1x+a_2x^2+a_3x^3+\cdots+a_nx^n \\ &=a_0+x\Bigg(a_1+x\Big(a_2+x\big(a_3+\cdots+x(a_{n-1}+xa_n)\big)\Big)\Bigg). \end{aligned} \tag{2.26}$$

To describe the algorithm iteratively, we introduce the following intermediate variables

$$y_k=\begin{cases} a_{n-1}+xa_n, & k=n, \\ a_{k-1}+xy_{k+1}, & k=n-1,n-2,\cdots,1. \end{cases}$$

Then we have $y_1=f(x)$. By implementing of y_k for each k, using the realizations formula given in Lemma 2.1, and stacking the implementations of n steps up, we obtain a σ_2 neural network with $\mathcal{O}(n)$ layers and where each layer has a constant width independent of n.

The second construction given in the following theorem can achieve same representation power with same amount of weights but much less layers.

Theorem 2.2. *If $f(x)$ is a polynomial of degree n on $\mathbb{R}$, then it can be represented exactly by a σ_2 neural network with $\lfloor\log_2 n\rfloor+1$ hidden layers, and the numbers of nodes and nonzero weights are both of order $\mathcal{O}(n)$. To be more precise, the number of nodes is bounded by $9n$, and number of nonzero weights is bounded by $61n$.*

Proof. Assume $f(x)=\sum_{j=0}^n a_jx^j$, $a_n\neq 0$. We first use an example with $n=15$ to demonstrate the process of network construction as follows:

$$\begin{aligned} f(x)&=a_{15}x^{15}+a_{14}x^{14}+\cdots+a_8x^8+a_7x^7+a_6x^6+\cdots+a_1x+a_0\\ &=\underbrace{x^8}_{\xi_{3,0}}\underbrace{\left\{\underbrace{x^4}_{\xi_{2,0}}\underbrace{\left[\underbrace{x^2}_{\xi_{1,0}}\underbrace{(a_{15}x+a_{14})}_{\xi_{1,8}}+\underbrace{(a_{13}x+a_{12})}_{\xi_{1,7}}\right]}_{\xi_{2,4}}+\underbrace{\left[x^2\underbrace{(a_{11}x+a_{10})}_{\xi_{1,6}}+\underbrace{(a_9x+a_8)}_{\xi_{1,5}}\right]}_{\xi_{2,3}}\right\}}_{\xi_{3,2}}\\ &\quad+\underbrace{\left\{x^4\underbrace{\left[x^2\underbrace{(a_7x+a_6)}_{\xi_{1,4}}+\underbrace{(a_5x+a_4)}_{\xi_{1,3}}\right]}_{\xi_{2,2}}+\underbrace{\left[x^2\underbrace{(a_3x+a_2)}_{\xi_{1,2}}+\underbrace{(a_1x+a_0)}_{\xi_{1,1}}\right]}_{\xi_{2,1}}\right\}}_{\xi_{3,1}}. \end{aligned} \tag{2.27}$$

Here ξ_{1,j_1}, $j_1=0,1,2,\cdots,8$, ξ_{2,j_2}, $j_2=0,1,2,\cdots,4$, and ξ_{3,j_3}, $j_3=0,1,2$ are the intermediate variable output of Layer 1, 2, 3, respectively. The final output is $f(x)=\xi_{3,0}\xi_{3,2}+\xi_{3,1}$.

We first describe the construction for the case $n\geq 4$ here.

Denote $m=\lfloor \log_2 n\rfloor$. We first extend $f(x)$ to include monomials up to degree $2^{m+1}-1$ by zero padding:

$$f(x):=\sum_{j=0}^{2^{m+1}-1} a_jx^j, \quad \text{where} \quad a_j=0, \quad \text{for } n+1\leq j\leq 2^{m+1}-1. \tag{2.28}$$

The process of building a σ_2 network to represent $f(x)$ is similar to the case $n=15$. We give details below.

1) The output of Layer 1 intermediate variables are:

$$\xi_{1,j}=a_{2j-1}x+a_{2j-2}=a_{2j-1}\beta_1^T\sigma_2(\omega_1x+\gamma_1)+a_{2j-2}, \quad j=1,2,\cdots,2^m, \tag{2.29}$$

$$\xi_{1,0}=x^2=\beta_2^T\sigma_2(\omega_2x), \tag{2.30}$$

which suggest

$$\boldsymbol{x}_1=\sigma_2\begin{pmatrix}\omega_1x+\gamma_1\\ \omega_2x\end{pmatrix}=\sigma_2(A_1x+b_1), \quad \text{where} \quad A_1=\begin{bmatrix}\omega_1\\ \omega_2\end{bmatrix}, \quad b_1=\begin{bmatrix}\gamma_1\\ \mathbf{0}\end{bmatrix}. \tag{2.31}$$

and

$$\boldsymbol{\xi}_1=A_{2,0}\boldsymbol{x}_1+b_{2,0}, \quad \text{where} \quad A_{2,0}=\begin{bmatrix}a_{21}\beta_1^T & \mathbf{0}\\ \mathbf{0} & \beta_2^T\end{bmatrix}, \quad b_{2,0}=\begin{bmatrix}a_{22}\\ 0\end{bmatrix}, \tag{2.32}$$

with $\boldsymbol{\xi}_1=[\xi_{1,1},\xi_{1,2},\cdots,\xi_{1,2^m},\xi_{1,0}]^T$, $a_{21}=[a_1,a_3,\cdots,a_{2^{m+1}-1}]^T$, $a_{22}=[a_0,a_2,\cdots,a_{2^{m+1}-2}]^T$.

2) The output of Layer 2 intermediate variables are:

$$\begin{aligned} \xi_{2,j} &= \xi_{1,0}\xi_{1,2j} + \xi_{1,2j-1} \\ &= \beta_1^T \sigma_2(\omega_1 \xi_{1,2j} + \gamma_1 \xi_{1,0}) + \beta_1^T \sigma_2\left(\omega_1 \xi_{1,2j-1} + \gamma_1\right), \quad j = 1,2,\cdots,2^{m-1}, && (2.33) \\ \xi_{2,0} &= (\xi_{1,0})^2 = \sigma_2(\xi_{1,0}), && (2.34) \end{aligned}$$

which imply

$$\boldsymbol{x}_2 = \sigma_2(A_{21}\boldsymbol{\xi}_1 + b_{21}), \quad \boldsymbol{x}_2, b_{21} \in \mathbb{R}^{(8\cdot 2^{m-1}+1)\times 1}, \ A_{21} \in \mathbb{R}^{(8\cdot 2^{m-1}+1)\times(2^m+1)}, \tag{2.35}$$

and most elements in A_{21}, b_{21} are zeros. The nonzero elements are given below using a Matlab subscript style as:

$$A_{21}(8j-8:8j,[2j-1,2j,2^m+1]) = \begin{bmatrix} \omega_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \omega_1 & \gamma_1 \end{bmatrix}, \quad b_{21}(8j-8:8j) = \begin{bmatrix} \gamma_1 \\ \mathbf{0} \end{bmatrix}, \tag{2.36}$$

for $j = 1,2,\cdots,2^{m-1}$, and the last element of $A_{2,1}$ is 1. According to the result (2.32) of Layer 1, we get

$$\boldsymbol{x}_2 = \sigma_2(A_2 \boldsymbol{x}_1 + b_2), \quad A_2 = A_{2,1}A_{2,0}, \quad b_2 = A_{2,1}b_{2,0} + b_{2,1}. \tag{2.37}$$

We also have

$$\boldsymbol{\xi}_2 = A_{3,0}\boldsymbol{x}_2, \quad \text{where} \quad A_{3,0} = \begin{bmatrix} I_{2^{m-1}} \otimes [\beta_1^T \ \beta_1^T] & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}, \tag{2.38}$$

Here $\boldsymbol{\xi}_2 = [\xi_{2,1}, \xi_{2,2}, \cdots, \xi_{2,2^{m-1}}, \xi_{2,0}]^T$, and $I_{2^{m-1}}$ is the identity matrix in $\mathbb{R}^{2^{m-1}}$. $\otimes$ stands for Kronecker product.

3) The output of Layer k $(3 \le k \le m)$ intermediate variables are:

$$\begin{aligned} \xi_{k,j} &= \xi_{k-1,0}\xi_{k-1,2j} + \xi_{k-1,2j-1} \qquad (j = 1,2,\cdots,2^{m-k+1}) \\ &= \beta_1^T \sigma_2(\omega_1 \xi_{k-1,2j} + \gamma_1 \xi_{k-1,0}) + \beta_1^T \sigma_2\left(\omega_1 \xi_{k-1,2j-1} + \gamma_1\right), && (2.39) \\ \xi_{k,0} &= (\xi_{k-1,0})^2 = \sigma_2(\xi_{k-1,0}). && (2.40) \end{aligned}$$

Denote $\boldsymbol{\xi}_k = [\xi_{k,1}, \xi_{k,2}, \cdots, \xi_{k,2^{m-k+1}}, \xi_{k,0}]^T$. We have

$$\boldsymbol{\xi}_k = A_{k+1,0}\boldsymbol{x}_k, \quad \boldsymbol{x}_k = \sigma_2(A_{k1}\boldsymbol{\xi}_{k-1} + b_{k1}), \tag{2.41}$$

where A_{k1}, b_{k1} have the same formula as A_{21}, b_{21} given in (2.36) except that the maximum value of j is 2^{m-k+1} rather than 2^{m-1}, and $A_{k+1,0}$ has the same formula as A_{30} given in (2.38) with $\mathbf{1}_{2^{m-1}\times 1}$ replaced by $\mathbf{1}_{2^{m-k+1}\times 1}$ and $\mathbf{1}_n = [1,\cdots,1]^T \in \mathbb{R}^{n\times 1}$. Combining (2.41) and (2.38), we get

$$\boldsymbol{x}_k = \sigma_2(A_k \boldsymbol{x}_{k-1} + b_k), \quad \text{where} \quad A_k = A_{k1}A_{k0}, \quad b_k = b_{k1}. \tag{2.42}$$

4) The output of Layer $m+1$ intermediate variables are:

$$\xi_{m+1,1}=\xi_{m,0}\xi_{m,2}+\xi_{m,1}=\beta_1^T\sigma_2(\omega_1\xi_{m,2}+\gamma_1\xi_{m,0})+\beta_1^T\sigma_2(\omega_1\xi_{m,1}+\gamma_1). \tag{2.43}$$

Written into the following form

$$\boldsymbol{\xi}_{m+1}:=[\xi_{m+1,1}]=A_{m+2,0}\boldsymbol{x}_{m+1}, \quad \boldsymbol{x}_{m+1}=\sigma_2(A_{m+1,1}\boldsymbol{\xi}_m+b_{m+1,1}), \tag{2.44}$$

we have

$$A_{m+1,1}=\begin{bmatrix}\omega_1 & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \omega_1 & \gamma_1\end{bmatrix}, \quad b_{m+1,1}=\begin{bmatrix}\gamma_1\\ \mathbf{0}\end{bmatrix}, \tag{2.45}$$

and

$$A_{m+2,0}=[\beta_1^T\ \beta_1^T], \quad b_{m+2,0}=0. \tag{2.46}$$

The iteration formula for $\boldsymbol{x}_{m+1}$ is $\boldsymbol{x}_{m+1}=\sigma_2(A_{m+1}\boldsymbol{x}_m+b_{m+1})$, where

$$A_{m+1}=A_{m+1,1}A_{m+1,0}, \quad b_{m+1}=b_{m+1,1}. \tag{2.47}$$

5) Since $\boldsymbol{\xi}_{m+1}=f(x)$, the network ends at Layer $m+2$, with $\boldsymbol{x}_{m+2}=\boldsymbol{\xi}_{m+1}$. So we get $A_{m+2}=A_{m+2,0}$, and $b_{m+2}=0$ from Eq. (2.44).

For $n<4$, the procedure can be obtained by removing some sub-steps from the cases $n\geq 4$. From the construction process, we see that the number of layers is $m+2$, the numbers of nodes in Layer 1 to Layer $m+1$ are 6, $8\times 2^{m-k+1}+1\,(2\leq k\leq m)$ and 8 respectively, and the number of nonzero weights in $\boldsymbol{A}_j$, $\boldsymbol{b}_j\,(1\leq j\leq m+2)$ are not bigger than 10, $(40\times 2^{m-1}+2)+8\times 2^{m-1}$, $(68\times 2^{m-j+1}+1)+4\times 2^{m-j+1}$ $(3\leq j\leq m)$, 72, 8 respectively. Summing up these numbers, we reach the desired bound. □

Remark 2.2. Theorem 2.1 says we can use a σ_2 network of scale $\mathcal{O}(\log_2 n)$ to represent x^n exactly. Theorem 2.2 says that any polynomial of degree less than n can be represented exactly by a σ_2 neural network with $\lfloor \log_2 n\rfloor+1$ hidden layers, and no more than $\mathcal{O}(n)$ nonzero weights. Such results are not available for ReLU network and neural networks using other non-polynomial activation functions, such as logistic, tanh, softplus, sinc etc. We note that the constants in the two theorems may not be optimal, but the orders of number of layers and number of nonzero weights are optimal.

2.1.2 Error bounds of approximating smooth functions by deep ReQU networks

Now we analyze the error of approximating general smooth functions using ReQU networks. We first introduce some notations and give a brief review of some classical results of polynomial approximation.

Let $\Omega\subseteq\mathbb{R}^d$ be the domain on which the function to be approximated is defined. For the 1-dimensional case in this section, we focus on $\Omega=I:=[-1,1]$. Similar discussions and results can be extended to $\Omega=[0,\infty)$ and $(-\infty,\infty)$ as well. We denote the set of polynomials with degree up to N defined on Ω by $P_N(\Omega)$, or simply P_N. Let $J_n^{\alpha,\beta}(x)$ be the Jacobi polynomial of degree n, $n\in\mathbb{N}_0$; the family of all these polynomials forms a complete set of orthogonal bases in the weighted $L^2_{\omega^{\alpha,\beta}}(I)$ space with respect to weight $\omega^{\alpha,\beta}(x)=(1-x)^\alpha(1+x)^\beta$ for $\alpha,\beta>-1$. To describe functions with high order regularity, we define the Jacobi-weighted Sobolev space $B^m_{\alpha,\beta}(I)$ as (see e.g. [54]):

$$B^m_{\alpha,\beta}(I):=\left\{u:\partial_x^k u\in L^2_{\omega^{\alpha+k,\beta+k}}(I),\ 0\le k\le m\right\},\quad m\in\mathbb{N},\tag{2.48}$$

with norm

$$\|f\|_{B^m_{\alpha,\beta}}:=\left(\sum_{k=0}^m\|\partial_x^k u\|^2_{L^2_{\omega^{\alpha+k,\beta+k}}}\right)^{1/2}.\tag{2.49}$$

Define the $L^2_{\omega^{\alpha,\beta}}$-orthogonal projection $\pi_N^{\alpha,\beta}$: $L^2_{\omega^{\alpha,\beta}}(I)\to P_N$ by requiring

$$\left(\pi_N^{\alpha,\beta}u-u,v\right)_{\omega^{\alpha,\beta}}=0,\quad \forall v\in P_N.\tag{2.50}$$

A detailed error estimate on the projection error $\pi_N^{\alpha,\beta}u-u$ is given in Theorem 3.35 of [54], by which we have the following theorem on the approximation error of ReQU networks.

Theorem 2.3. *Let $\alpha,\beta>-1$, $N\ge1$. For any $u\in B^m_{\alpha,\beta}(I)$, there exist a ReQU network Φ_N^u with $\lfloor\log_2 N\rfloor+1$ hidden layers, $\mathcal{O}(N)$ nodes, and $\mathcal{O}(N)$ nonzero weights, satisfying the following estimates.*

1) If $0\le l\le m\le N+1$, we have

$$\left\|\partial_x^l\left(R_{\sigma_2}(\Phi_N^u)-u\right)\right\|_{\omega^{\alpha+l,\beta+l}}\le c\sqrt{\frac{(N-m+1)!}{(N-l+1)!}}(N+m)^{(l-m)/2}\|\partial_x^m u\|_{\omega^{\alpha+m,\beta+m}}.\tag{2.51}$$

2) If $m>N+1\ge l$, we have

$$\left\|\partial_x^l\left(R_{\sigma_2}(\Phi_N^u)-u\right)\right\|_{\omega^{\alpha+l,\beta+l}}\le c(2\pi N)^{-1/4}\left(\frac{\sqrt{e/2}}{N}\right)^{N-l+1}\|\partial_x^{N+1}u\|_{\omega^{\alpha+N+1,\beta+N+1}}.\tag{2.52}$$

Here $c\approx1$ for $N\gg1$.

Proof. For any given $u\in B^m_{\alpha,\beta}(I)$, the polynomial $f=\pi_N^{\alpha,\beta}u\in P_N$. The projection error $\pi_N^{\alpha,\beta}u-u$ is estimated by Theorem 3.35 in [54], which is (2.51) and (2.52) with $R_{\sigma_2}(\Phi_N^u)$ replaced by $\pi_N^{\alpha,\beta}u$. By Theorem 2.2, f can be represented exactly by a ReQU network Φ_N^u with $\lfloor\log_2 N\rfloor+1$ hidden layers, $\mathcal{O}(N)$ nodes, and $\mathcal{O}(N)$ nonzero weights, i.e. $R_{\sigma_2}(\Phi_N^u)=\pi_N^{\alpha,\beta}u$. We thus obtain estimation (2.51) and (2.52). □

Remark 2.3. In (2.51) and (2.52), we allow the error measured in high-order derivatives, i.e. $l\geq 3$, because $R_{\sigma_2}(\Phi^u_N)$ is an exact realization of a polynomial, which is infinitely differentiable. In practice, if Φ^u_N is a trained network with numerical error, we can not measure the error with derivatives order ≥ 3, since $\partial^3_x\sigma_2(x)$ is not in L^2 space.

Based on Theorem 2.3, we can analyze the network complexity of ε-approximation of a given function with certain smoothness. For simplicity, we only consider the case with $l=0$. The result is given in the following theorem.

Theorem 2.4. *For any given function $f(x)\in B^m_{\alpha,\beta}(I)$ with norm less than* 1*, where m is either a fixed positive integer or infinity, and for $\varepsilon\in(0,1)$ small enough, there exists a ReQU network Φ^f_ε with number of layers L, number of nonzero weights N satisfying*

- *if m is a fixed positive integer, then $L=\mathcal{O}\left(\frac{1}{m}\log_2\frac{1}{\varepsilon}\right)$, and $N=\mathcal{O}\left(\varepsilon^{-\frac{1}{m}}\right)$;*
- *if $m=\infty$, i.e. $f(x)\in B^\infty_{\alpha,\beta}(I)$, then $L=\mathcal{O}\left(\log_2\left(\log\frac{1}{\varepsilon}\right)\right)$, and $N=\mathcal{O}\left(\frac{\log(1/\varepsilon)}{\log_2(\log(1/\varepsilon))}\right)$,*

that approximates f within an error tolerance ε, i.e.

$$\|R_{\sigma_2}(\Phi^f_\varepsilon)-f\|_{\omega^{\alpha,\beta}(I)}\leq\varepsilon. \tag{2.53}$$

Proof. For a fixed m, or $N\gg m$, we obtain from (2.51) that

$$\|R_{\sigma_2}(\Phi^u_N)-u\|_{\omega^{\alpha,\beta}(I)}\leq cN^{-m}\|\partial^m_x u\|_{\omega^{\alpha+m,\beta+m}}. \tag{2.54}$$

By above estimate, we obtain that to achieve an error tolerance ε to approximate a function with $B^m_{\alpha,\beta}(I)$ norm less than 1, it suffices to take $N=\left(\frac{c}{\varepsilon}\right)^{\frac{1}{m}}$. For fixed m, we have $N=\mathcal{O}\left(\varepsilon^{-\frac{1}{m}}\right)$, the depth of the corresponding ReQU network is $L=\mathcal{O}\left(\frac{1}{m}\log_2\frac{1}{\varepsilon}\right)$.

For $f\in B^\infty_{\alpha,\beta}$, by taking $m=\infty$ in Theorem 2.3, we have

$$\|R_{\sigma_2}(\Phi^u_N)-u\|_{\omega^{\alpha,\beta}(I)}\leq c(2\pi N)^{-\frac{1}{4}}\left(\frac{\sqrt{e/2}}{N}\right)^{N+1}\|u\|_{B^\infty_{\alpha,\beta}}\leq c'e^{-\gamma N}\|u\|_{B^\infty_{\alpha,\beta}}, \tag{2.55}$$

where c' is a general constant, and $\gamma\approx\mathcal{O}(\log N)$ can be larger than any fixed positive number for sufficient large N. To approximate a function with $B^\infty_{\alpha,\beta}(I)$ norm less than 1 with error $\varepsilon=c'e^{-\gamma N}$, it suffices to take $N=\frac{1}{\gamma}\log\left(\frac{c'}{\varepsilon}\right)$, which means $N=\mathcal{O}\left(\frac{\log(1/\varepsilon)}{\log_2(\log(1/\varepsilon))}\right)$. The depth of the corresponding ReQU network is $L=\mathcal{O}\left(\log_2\left(\log\frac{1}{\varepsilon}\right)\right)$. Here ε is assumed to be small enough such that $\log_2\left(\log\frac{c'}{\varepsilon}\right)$ is no less than 1. □

2.2 Approximation by deep networks using general rectified power units

The results of approximation monomials, polynomials and general smooth functions by ReQU networks discussed in Subsection 2.1 can be extended to general RePU networks.

To keep the paper short, we only present the results on approximating monomials with RePU in Theorem 2.5. The other results similar to ReQU networks can be obtained but the details are quite lengthy, we report them in a separate paper [55].

Theorem 2.5. *Regarding the problem of using $\sigma_s(x)$ $(2\le s\in\mathbb{N})$ neural networks to exactly represent monomial x^n, $n\in\mathbb{N}$, we have the following results:*

(1) If $s=n$, the monomial x^n can be realized exactly using a σ_s networks having only 1 hidden layer with two nodes.

(2) If $1\le n<s$, the monomial x^n can be realized exactly using a σ_s networks having only 1 hidden layer with no more than $2s$ nodes.

(3) If $n>s\ge 2$, the monomial x^n can be realized exactly using a σ_s networks having $\lfloor \log_s n\rfloor+2$ hidden layers with no more than $(6s+2)(\lfloor\log_s n\rfloor+2)$ nodes, no more than $\mathcal{O}(25s^2\lfloor\log_s n\rfloor)$ nonzero weights.

Proof. (1) It is easy to check that x^s has an exact σ_s realization given by

$$\rho_s(x):=\sigma_s(x)+(-1)^s\sigma_s(-x)=x^s. \tag{2.56}$$

(2) For the case of $1\le n<s$, we consider the following linear combination

$$a_0+\sum_{k=1}^{s}a_k\rho_s(x+b_k)=a_0+\sum_{k=1}^{s}a_k\left(\sum_{j=0}^{s}C_j^s b_k^{s-j}x^j\right)=a_0+\sum_{j=0}^{s}C_j^s\left(\sum_{k=1}^{s}a_k b_k^{s-j}\right)x^j, \tag{2.57}$$

where a_0,a_k,b_k, $k=1,\cdots,s$ are parameters to be determined. C_j^s are binomial coefficients. The above expression is equal to x^n, provided that the parameters solve the following linear system:

$$D_{s+1}\boldsymbol{a}:=\begin{bmatrix}1 & 1 & \cdots & 1 & 0\\ \vdots & \vdots & & \vdots & \vdots\\ b_1^{s-n} & b_2^{s-n} & \cdots & b_s^{s-n} & 0\\ \vdots & \vdots & & \vdots & \vdots\\ b_1^{s-1} & b_2^{s-1} & \cdots & b_s^{s-1} & 0\\ b_1^s & b_2^s & \cdots & b_s^s & 1\end{bmatrix}\begin{bmatrix}a_1\\ \vdots\\ \cdot\\ a_s\\ a_0\end{bmatrix}=\begin{bmatrix}0\\ \vdots\\ (C_n^s)^{-1}\\ \vdots\\ 0\end{bmatrix}, \tag{2.58}$$

where the top-left $s\times s$ submatrix of D_{s+1} is a Vandermonde matrix, which is invertible as long as b_k, $k=1,\cdots,s$ are distinct. For simplicity, we choose b_k, $k=0,\cdots,s$ to be equidistant points, then (2.58) is uniquely solvable. Solving for $a_0,\cdots,a_s$ we obtain an exact representation of x^n using (2.57), which corresponds to a neural network having one hidden layer with no more than $2s$ σ_s units.

For example, when $s=2$, we may take $b_1=-1$, $b_1=1$. Solving Eq. (2.58) with $n=1$, we get $a_1=-\frac{1}{4}$, $a_2=\frac{1}{4}$, and $a_0=0$. Thus

$$x=\frac{1}{4}\rho_2(x+1)-\frac{1}{4}\rho_2(x-1).$$

When $s=3$, take $b_1=-1$, $b_2=0$, $b_3=1$, we obtain

$$\begin{aligned} x&=\frac{1}{6}\left[\rho_3(x-1)-2\rho_3(x)+\rho_3(x+1)\right],\\ x^2&=\frac{1}{6}\left[\rho_3(x+1)-\rho_3(x-1)\right]-\frac{1}{3}. \end{aligned}$$

(3) Now, we consider the case $n>s\geq 2$, $n\in\mathbb{N}$. For any given numbers $y,z\in\mathbb{R}$, using the identity

$$yz=\frac{1}{4}\left[(y+z)^2-(y-z)^2\right] \tag{2.59}$$

and the fact that $(y+z)^2$, $(y-z)^2$ both can be realized exactly by a one layer σ_s network with no more than $2s$ nodes, we conclude that the product yz can be realized by one layer σ_s network with no more than $4s$ nodes. To realize x^n by σ_s, we rewrite n in the following form

$$n=a_m\cdot s^m+a_{m-1}\cdot s^{m-1}+\cdots+a_1\cdot s+a_0, \quad m=\lfloor \log_s m\rfloor, \tag{2.60}$$

where $a_j\in\{0,1,\cdots,s-1\}$ for $j=0,1,\cdots,m-1$ and $a_m=1$. So we have

$$x^n=(x^{s^m})^{a_m}(x^{s^{m-1}})^{a_{m-1}}\cdots(x^s)^{a_1}(x)^{a_0}. \tag{2.61}$$

Define $\xi_k=x^{s^k}$, $z_{k+1}=(\xi_k)^{a_k}$, $k=0,1,\cdots,m$, and

$$y_2=x^{a_0}, \quad y_{k+2}=z_{k+1}y_{k+1}\ \left(=(x^{s^k})^{a_k}y_{k+1}\right), \quad \text{for } k=1,\cdots,m, \tag{2.62}$$

we have $y_{m+2}=x^n$. Eq. (2.62) can be regarded as an iteration scheme, with iteration variables ξ_k,y_k,z_k, where the subscript k stands for the iteration step. A schematic diagram for this iteration is given in Fig. 2. Different to Theorem 2.1, for $s>2$, we need a deep σ_s neural network with $m+2$ hidden layers to realize $x^n,n>s$, due to the introduction of intermediate variables z_k. In each layer, we need no more than $2+2s+4s$ activation nodes to calculate $\xi_{k+1}=\rho_s(\xi_k)$, $z_{k+1}=(\xi_k)^{a_k}$, and $y_{k+1}=z_ky_k$. So in total we need no more than $(6s+2)(m+2)=\mathcal{O}(6s\log_s n)$ nodes. A direct calculation shows that the number of nonzero weights in the network is no more than $\mathcal{O}(25s^2\log_s n)$. The theorem is proved. □

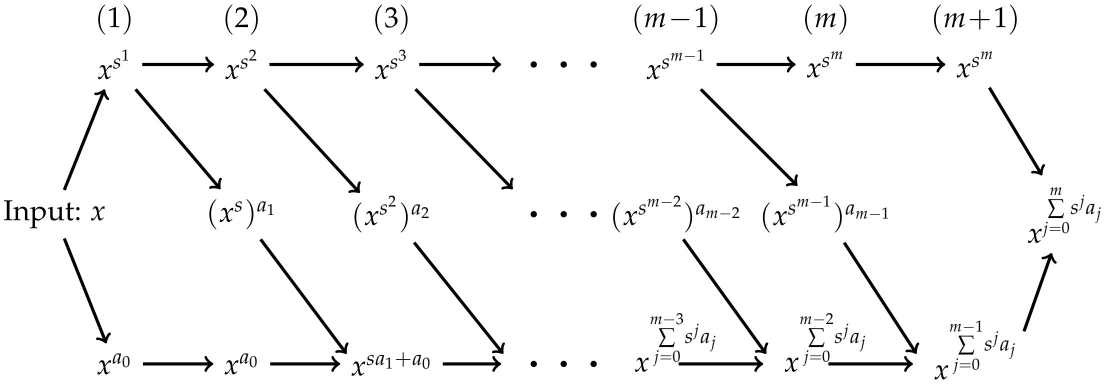
(1)
(2)
(3)
(m−1)
(m)
(m+1)
x^{s^1}
x^{s^2}
x^{s^3}
$x^{s^{m-1}}$
x^{s^m}
x^{s^m}
Input: x
$(x^s)^{a_1}$
$(x^{s^2})^{a_2}$
$(x^{s^{m-2}})^{a_{m-2}}$
$(x^{s^{m-1}})^{a_{m-1}}$
$x^{\sum_{j=0}^{m} s^j a_j}$
x^{a_0}
x^{a_0}
$x^{sa_1+a_0}$
$x^{\sum_{j=0}^{m-3} s^j a_j}$
$x^{\sum_{j=0}^{m-2} s^j a_j}$
$x^{\sum_{j=0}^{m-1} s^j a_j}$

Figure 2: A schematic diagram for σ_s network realization of x^n, $n>s$. (j) represents the j-th hidden layer of intermediate variables.

3 Approximation of multivariate functions

In this section, we discuss how to approximate multivariate smooth functions by ReQU networks. Similar to the univariate case, we first study the representation of polynomials then discuss the approximation error of general smooth functions.

3.1 Deep ReQU network representations of multivariate polynomials

Theorem 3.1. *If $f(x)$ is a multivariate polynomial with* total *degree n on $\mathbb{R}^d$, then there exists a σ_2 neural network having $d\lfloor\log_2 n\rfloor+d$ hidden layers with no more than $\mathcal{O}(C_d^{n+d})$ activation functions and nonzero weights, that can represent f with no error. We note that, here the constant behind the big $\mathcal{O}$ can be bounded independent of d.*

Proof. 1) We first consider the 2-dimensional case. Suppose $f(x,y)=\sum_{i+j=0}^{n} a_{ij}x^i y^j$ and $n\ge 4$ (the results for $n\le 3$ are similar but easier, so skipped here). To represent $f(x,y)$ exactly with a σ_2 neural network based on the results for the 1-dimensional case given in Theorem 2.2, we first rewrite $f(x,y)$ as

$$f(x,y)=\sum_{i=0}^{n}\left(\sum_{j=0}^{n-i} a_{ij}y^j\right)x^i=:\sum_{i=0}^{n} a_i^y x^i, \quad \text{where} \quad a_i^y=\sum_{j=0}^{n-i} a_{ij}y^j. \tag{3.1}$$

So to realize $f(x,y)$, we can first realize a_i^y, $i=0,\cdots,n-1$ using n small σ_2 networks Φ_i, $i=0,\cdots,n-1$, i.e. $R_{\sigma_2}(\Phi_i)(y)=a_i^y$ for given input y; then use a σ_2 network Φ_n to realize the 1-dimensional polynomials $f(x,y)=\sum_{i=0}^{n} a_i^y x^i$. There are two places that need some technical treatment, the details are given below.

(1) The network Φ_n takes a_i^y, $i=0,\cdots,n$ and x as input. So these quantities must be presented at the same layer of the overall neural network, because we do not want connections over non-adjacent layers. By Theorem 2.2, the largest depth of networks

Φ_i, $i=0,\cdots,n-1$ is $\lfloor \log_2 n \rfloor+2$, so we can lift x to layer $\lfloor \log_2 n \rfloor+2$ using multiple $id_{\mathbb{R}}(\cdot)$ operations. Similarly, we also keep a record of input y in each layer using multiple $id_{\mathbb{R}}(\cdot)$ operations, such that Φ_i, $i=1,\cdots,n-1$ can start from appropriate layer and generate output exactly at layer $\lfloor \log_2 n \rfloor+2$. The overall cost for recording x,y in layers $1,\cdots,\lfloor \log_2 n \rfloor+2$ is $\mathcal{O}(\lfloor \log_2 n \rfloor+2)$, which is small comparing to the number of coefficients C_2^{n+2}.

(2) While realizing $\sum_{i=0}^n a_i^y x^i$, the coefficients a_i^y, $i=0,\cdots,n$ are network input instead of fixed parameters. So when applying the network construction given in Theorem 2.2, we need to modify the structure of the first layer of the network. More precisely, Eq. (2.29) in Theorem 2.2 should be changed to

$$\begin{aligned}\xi_{1,j}^y &= a_{2j-1}^y x + a_{2j-2}^y \\ &= \beta_1^T \sigma_2\left(\omega_1 x + \gamma_1 a_{2j-1}^y\right) + \beta_1^T \sigma_2\left(\omega_1 a_{2j-2}^y + \gamma_1\right), \quad j=1,\cdots,2^m. \end{aligned} \tag{3.2}$$

So the number of nodes for the first layer changed from 6 to $2+8\cdot 2^m$, the number of nonzero weights for the first layer changed from 10 to $16\cdot 2^m+2$. So the number of hidden layers, number of nodes and nonzero weights of Φ_n can be bounded by $\lfloor \log_2 n \rfloor+1$, $17n$, and $77n$ respectively.

Assembling $\Phi_0,\cdots,\Phi_n$, the overall network to represent $f(x,y)$ has $2\lfloor \log_2 n \rfloor+3$ layers with number of nodes no more than

$$\sum_{j=0}^{n-1} 9(n-j)+17n+8(m+2) = 9\frac{n(n+1)}{2}+17n+8m+16 = \mathcal{O}(C_d^{n+d}),$$

and number of weights no more than

$$\sum_{j=0}^{n-1} 61(n-j)+77n+16(m+2)\times 2+12n = 61\frac{n(n+1)}{2}+89n+32m+64 = \mathcal{O}(C_d^{n+d}).$$

Thus, we proved that the theorem is true for the case $d=2$.

2) The case $d>2$ can be proved by mathematical induction using the similar procedure as done for $d=2$ case. Note that we pad in some zeros in each direction in the iteration. Since after each dimension iteration, the number of degree of freedom are geometrically reduced, by a straightforward calculation, one can show that the constant behind the big $\mathcal{O}$ can be made independent of dimension d. An improved algorithm using less padding zeros is proposed in another paper [55]. □

Using a similar approach as in Theorem 3.1, one can easily prove the following theorem.

Theorem 3.2. *For a polynomial f_N in a tensor product space $Q_N^d(I_1\times\cdots\times I_d) := P_N(I_1)\otimes\cdots\otimes P_N(I_d)$, there exists a σ_2 network having $d\lfloor \log_2 N \rfloor+d$ hidden layers with no more than $\mathcal{O}(N^d)$ activation functions and nonzero weights, can represent f_N with no error.*

3.2 Error bounds of approximating multivariate functions by ReQU networks

Now we analyze the error of approximating general multivariate smooth functions using ReQU networks.

For a vector $\boldsymbol{x}=(x_1,\cdots,x_d)\in\mathbb{R}^d$, we define $|\boldsymbol{x}|_1:=|x_1|+\cdots+|x_d|$, $|\boldsymbol{x}|_\infty:=\max_{i=1}^d|x_i|$. Define the high dimensional Jacobi weight as $\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}(\boldsymbol{x}):=\omega^{\alpha_1,\beta_1}(x_1)\cdots\omega^{\alpha_d,\beta_d}(x_d)$. We define the multidimensional Jacobi-weighted Sobolev space $B^m_{\alpha,\beta}(I^d)$ as [54]:

$$B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d):=\Big\{u\in L^2(I^d)\,|\,\partial^{\boldsymbol{k}}_{\boldsymbol{x}}u:=\partial^{k_1}_{x_1}\cdots\partial^{k_d}_{x_d}u\in L^2_{\omega^{\boldsymbol{\alpha}+\boldsymbol{k},\boldsymbol{\beta}+\boldsymbol{k}}}(I^d),\ \boldsymbol{k}\in\mathbb{N}^d_0,\ |\boldsymbol{k}|_1\le m\Big\},\quad m\in\mathbb{N}_0,$$

with norm and semi-norm

$$\|u\|_{B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}:=\Big(\sum_{|\boldsymbol{k}|_1\le m}\Big\|\partial^{\boldsymbol{k}}_{\boldsymbol{x}}u\Big\|^2_{L^2_{\omega^{\boldsymbol{\alpha}+\boldsymbol{k},\boldsymbol{\beta}+\boldsymbol{k}}}}\Big)^{1/2},\quad |u|_{B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}:=\Big(\sum_{|\boldsymbol{k}|_1=m}\Big\|\partial^{\boldsymbol{k}}_{\boldsymbol{x}}u\Big\|^2_{L^2_{\omega^{\boldsymbol{\alpha}+\boldsymbol{k},\boldsymbol{\beta}+\boldsymbol{k}}}}\Big)^{1/2}.$$

Define the $L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}$-orthogonal projection $\pi^{\boldsymbol{\alpha},\boldsymbol{\beta}}_N$: $L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}(I^d)\to Q^d_N(I^d)$ by the property

$$\Big(\pi^{\boldsymbol{\alpha},\boldsymbol{\beta}}_N u-u,v\Big)_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}=0,\quad \forall v\in Q^d_N(I^d). \tag{3.3}$$

Then for $u\in B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, we have the following error estimate (see Theorem 8.1 and Remark 8.13 in [54]):

$$\|\pi^{\boldsymbol{\alpha},\boldsymbol{\beta}}_N u-u\|_{L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}(I^d)}\le cN^{-m}|u|_{B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}},\quad 1\le m\le N, \tag{3.4}$$

where c is an absolute constant. Combining (3.4) and Theorem 3.2, we obtain the following upper bound for the ε-approximation of functions in $B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$ space.

Theorem 3.3. *For any $u\in B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, with $|u|_{B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)}\le 1$, $\boldsymbol{\alpha},\boldsymbol{\beta}\in(-1,\infty)^d$, and any $\varepsilon\in(0,1)$ there exists a σ_2 neural network Φ^u_ε having $\mathcal{O}\big(\frac{d}{m}\log_2\frac{1}{\varepsilon}+d\big)$ layers with no more than $\mathcal{O}\big(\varepsilon^{-d/m}\big)$ nodes and nonzero weights, that approximates u with $L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}(I^d)$-error less than ε, i.e.*

$$\|R_{\sigma_2}(\Phi^u_\varepsilon)-u\|_{L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}(I^d)}\le\varepsilon. \tag{3.5}$$

Remark 3.1. According to the classic nonlinear approximation theory by DeVore, Howard and Micchelli [56], the results of Theorem 2.4 (first part) and Theorem 3.3 are optimal in the case that the approximation depends on the function to be approximated continuously.

Remark 3.2. Note that results for approximating functions in weighted Sobolev space given in Theorem 3.3 can be extended to C^k if k is sufficient large, similar to the second part of Theorem 2.4. Comparing this result with Theorem 1 in [21], we see that the number of computational units and nonzero weights needed by a ReQU network to approximate a function $u\in B^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$ for m sufficient large, with an error tolerance ε is less than that needed by a ReLU network. The ReLU network is $\log\frac{1}{\varepsilon}$ times larger than corresponding ReQU network. For low accuracy approximation, the factor $\mathcal{O}(\log\frac{1}{\varepsilon})$ is not very big, but for high accuracy approximations, this factor can be as large as several dozens, which could make a big difference in large scale computations.

Note that, for functions with fixed lower order continuity, ReLU network can give good approximation using less number of layers, or use very deep ReLU networks to break the bounds given in Theorem 3.3. We refer interested readers to the recent works by Voigtlaender and Petersen [57], and Yarotsky [58].

4 High-dimensional functions with sparse polynomial approximations

In last section, we showed that for a d-dimensional function with partial derivatives up to order m in $L^2(I^d)$ can be approximated within error ε by a ReQU neural network with complexity $\mathcal{O}(\varepsilon^{-d/m})$. When m is fixed or much smaller than d, the network complexity has an exponential dependence on d. However, in a lot of applications, high-dimensional problems may have low intrinsic dimension (see e.g. [59, 60]). One particular example are high-dimensional tensor product functions(or linear combinations of finite terms of tensor product functions), which can be well approximated by a *hyperbolic cross* or *sparse grid* truncated series.

4.1 A brief review of hyperbolic cross approximations and sparse grids

Sparse grids were originally introduced by S. A. Smolyak [30] to integrate or interpolate high dimensional functions. Hyperbolic cross approximation is a technique similar to sparse grids but without the concept of grids. We introduce hyperbolic cross approximation by considering a tensor product function: $f(\boldsymbol{x})=f_1(x_1)\cdots f_d(x_d)$. Suppose that $f_1,\cdots,f_d$ have similar regularity that can be well approximated by using an orthonormal bases $\{\phi_k,\, k=0,1,\cdots.\}$; that is,

$$f_i(x)=\sum_{k=0}^{\infty} b_k^{(i)}\phi_k(x), \quad |b_k^{(i)}|\le c\bar{k}^{-r}, \quad i=1,\cdots,d,$$

where c is a general constant, $r\ge 1$ is a constant depending on the regularity of f_i, $\bar{k}:=\max\{1,k\}$. So we have an expansion for f as

$$f(\boldsymbol{x})=\prod_{i=1}^{d}\left(\sum_{k=0}^{\infty} b_k^{(i)}\phi_k(x_i)\right)=\sum_{\boldsymbol{k}\in\mathbb{N}_0^d} b_{\boldsymbol{k}}\phi_{\boldsymbol{k}}(\boldsymbol{x}),$$

where

$$|b_{\boldsymbol{k}}|=|b_{k_1}^{(1)}\cdots b_{k_d}^{(d)}|\le c^d(\bar{k}_1\cdots\bar{k}_d)^{-r}, \quad \phi_{\boldsymbol{k}}(\boldsymbol{x})=\phi_1(x_1)\cdots\phi_d(x_d).$$

Thus, to have a best approximation of $f(\boldsymbol{x})$ using finite terms, one should take

$$f_N := \sum_{\boldsymbol{k}\in\chi_N^d} b_{\boldsymbol{k}}\phi_{\boldsymbol{k}}(\boldsymbol{x}), \tag{4.1}$$

where

$$\chi_N^d := \Big\{\boldsymbol{k}=(k_1,\cdots,k_d)\in\mathbb{N}_0^d \,|\, \bar{k}_1\cdots\bar{k}_d \le N\Big\} \tag{4.2}$$

is the hyperbolic cross index set. We call f_N defined by (4.1) a hyperbolic cross approximation of f.

For general functions defined on I^d, we choose $\phi_{\boldsymbol{k}}$ to be multivariate Jacobi polynomials $J_{\boldsymbol{n}}^{\boldsymbol{\alpha},\boldsymbol{\beta}}$, and define the hyperbolic cross polynomial space as

$$X_N^d := \text{span}\{J_{\boldsymbol{n}}^{\boldsymbol{\alpha},\boldsymbol{\beta}},\ \boldsymbol{n}\in\chi_N^d\}. \tag{4.3}$$

Note that the definition of X_N^d doesn't depend $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. $\{J_{\boldsymbol{n}}^{\boldsymbol{\alpha},\boldsymbol{\beta}}\}$ is used to served as a set of bases for X_N^d. To study the error of hyperbolic cross approximation, we define Jacobi-weighted Korobov-type space

$$\mathcal{K}_{\boldsymbol{\alpha},\boldsymbol{\beta}}^m(I^d) := \Big\{u\in L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}(I^d)\ :\ \partial_{\boldsymbol{x}}^{\boldsymbol{k}}u\in L^2_{\omega^{\boldsymbol{\alpha}+\boldsymbol{k},\boldsymbol{\beta}+\boldsymbol{k}}}(I^d),\ 0\le\ |\boldsymbol{k}|_\infty \le m\Big\}, \quad \text{for } m\in\mathbb{N}_0, \tag{4.4}$$

with norm and semi-norm

$$\|u\|_{\mathcal{K}_{\boldsymbol{\alpha},\boldsymbol{\beta}}^m} := \Bigg(\sum_{|\boldsymbol{k}|_\infty\le m}\Big\|\partial_{\boldsymbol{x}}^{\boldsymbol{k}}u\Big\|^2_{L^2_{\omega^{\boldsymbol{\alpha}+\boldsymbol{k},\boldsymbol{\beta}+\boldsymbol{k}}}}\Bigg)^{\frac{1}{2}}, \quad |u|_{\mathcal{K}_{\boldsymbol{\alpha},\boldsymbol{\beta}}^m} := \Bigg(\sum_{|\boldsymbol{k}|_\infty = m}\Big\|\partial_{\boldsymbol{x}}^{\boldsymbol{k}}u\Big\|^2_{L^2_{\omega^{\boldsymbol{\alpha}+\boldsymbol{k},\boldsymbol{\beta}+\boldsymbol{k}}}}\Bigg)^{\frac{1}{2}}. \tag{4.5}$$

For any given $u\in\mathcal{K}^0_{\boldsymbol{\alpha},\boldsymbol{\beta}}(=B^0_{\boldsymbol{\alpha},\boldsymbol{\beta}})$, the hyperbolic cross approximation $\pi_{N,H}^{\boldsymbol{\alpha},\boldsymbol{\beta}}u\in X_N^d$ can be defined as a projection by requiring

$$(\pi_{N,H}^{\boldsymbol{\alpha},\boldsymbol{\beta}}u-u,v)_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}=0, \quad \forall v\in X_N^d. \tag{4.6}$$

Then we have the following error estimate about the hyperbolic cross approximation (see Theorem 2.2 in [33]):

$$\|\partial_{\boldsymbol{x}}^{\boldsymbol{l}}(\pi_{N,H}^{\boldsymbol{\alpha},\boldsymbol{\beta}}u-u)\|_{\omega^{\boldsymbol{\alpha}+\boldsymbol{l},\boldsymbol{\beta}+\boldsymbol{l}}} \le D_1 N^{|\boldsymbol{l}|_\infty - m}|u|_{\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}, \quad 0\le\boldsymbol{l}\le m,\ m\ge 1, \tag{4.7}$$

where D_1 is a constant independent of N. It is known that the cardinality of χ_N^d is of order $\mathcal{O}(N(\log N)^{d-1})$ in [33]. The above error estimate says that to approximate a function $u\in\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ with an error tolerance ε, one only needs a space of Jacobi polynomials of dimension at most $\mathcal{O}\big(\varepsilon^{-1/m}(\frac{1}{m}\log\frac{1}{\varepsilon})^{d-1}\big)$, the exponential dependence on d is weakened (cp. Theorem 3.3). To remove the exponential term $(\log\frac{1}{\varepsilon})^{d-1}$, one may consider a more general sparse polynomial space [33]:

$$X^d_{N,\gamma} := \text{span}\{J_{\boldsymbol{n}}^{\boldsymbol{\alpha},\boldsymbol{\beta}},\ (\Pi_{i=1}^d\bar{n}_i)|\boldsymbol{n}|_\infty^{-\gamma}\le N^{1-\gamma}\}, \quad -\infty\le\gamma<1. \tag{4.8}$$

In particular, $X^d_{N,0} = X^d_N$ is the hyperbolic cross space defined in (4.3), and $X^d_{N,-\infty} := \operatorname{span}\{J^{\boldsymbol{\alpha},\boldsymbol{\beta}}_{\boldsymbol{n}}, |\boldsymbol{n}|_\infty \leq N\}$ is the standard full grid. For $0<\gamma<1$, it is known that (see lemma 3 in [32]):

$$\operatorname{Card}(X^d_{N,\gamma}) = C(\gamma,d)N, \quad 0<\gamma<1, \tag{4.9}$$

where $C(\gamma,d)$ is a constant that depends on γ and d but is independent of N. We call $X^d_{N,\gamma}, 0<\gamma<1$ optimized hyperbolic cross polynomial space. It is proved by Shen and Wang that the $L^2_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}$-orthogonal projection $\pi^{\boldsymbol{\alpha},\boldsymbol{\beta}}_{N,\gamma}$ from Korobov space to $X^d_{N,\gamma}$ satisfies the following estimate (see Theorem 2.3 in [33]):

$$\|\pi^{\boldsymbol{\alpha},\boldsymbol{\beta}}_{N,\gamma} u - u\|_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}} \leq D_2 N^{-m(1-\gamma(1-\frac{1}{d}))} |u|_{\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}, \quad 0<\gamma<1, \tag{4.10}$$

where D_2 is a constant independent of N. From (4.9) and (4.10), we get that to approximate a function $u \in \mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ with an error tolerance ε, one only needs a space of Jacobi polynomials of dimension at most $\mathcal{O}\big(\varepsilon^{-1/m(1-\gamma(1-\frac{1}{d}))}\big)$. We will later use this estimate to derive another upper bound of approximating functions in $\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}$ using deep ReQU networks.

In practice, the exact hyperbolic cross projection is not easy to calculate. An alternate approach is the sparse grid, which uses hierarchical interpolation schemes to build a hyperbolic cross-like approximation of high dimensional functions. To define sparse grids for I^d, we first define the underlying 1-dimensional interpolations. Given a series of interpolation point sets $\mathcal{X}^i = \{x^i_1, \cdots, x^i_{m_i}\} \subseteq [-1,1]$, $m_i = \operatorname{Card}(\mathcal{X}^i)$, $i=1,2,\cdots$, with $0<m_1<m_2<\cdots$, the interpolation on $\mathcal{X}^i$ for $f \in C^0(I)$ is defined as

$$\mathcal{U}^i(f) = \sum_{j=1}^{m_i} f(x^i_j)\ell^i_j(x), \tag{4.11}$$

where $\ell^i_j(x) \in P_{m_i-1}([-1,1])$ $(j=1,2,\cdots,m_i)$ are the Lagrange interpolation polynomials for the interpolation points $\mathcal{X}^i$. The sparse grid interpolation for high-dimension function $f \in C^0(I^d)$ is defined as [30]:

$$\mathcal{A}(q,d)(f) = \sum_{d \leq |\boldsymbol{i}|_1 \leq q} \Big(\Delta^{i_1} \otimes \cdots \otimes \Delta^{i_d}\Big)(f), \quad q \geq d, \tag{4.12}$$

where $\Delta^i = \mathcal{U}^i - \mathcal{U}^{i-1}$, $i \in \mathbb{N}$. For convenience, we define $\mathcal{U}^0 := 0$, $m_0 = 0$, $\mathcal{X}^0 = \emptyset$. Formally, (4.12) can be defined on any grids $\{\mathcal{X}^i, i=1,2,\cdots,q-d+1\}$. However, to have a one-to-one transform between the values on interpolation points and the coefficients of linearly independent bases in the interpolation space, we need $\{\mathcal{X}^i, i=1,2,\cdots,q-d+1\}$ to be nested, i.e. $\mathcal{X}^1 \subset \mathcal{X}^2 \subset \cdots \subset \mathcal{X}^{q-d+1}$. Fast transforms between physical values and interpolation coefficients always exist for sparse grid interpolations using nested grids [40,41]. Define sparse grid index set as

$$\mathcal{I}^q_d := \bigcup_{d \leq |\boldsymbol{i}|_1 \leq q} \tilde{\mathcal{I}}^{i_1} \times \cdots \times \tilde{\mathcal{I}}^{i_d}, \quad \text{where } \tilde{\mathcal{I}}^k := \mathcal{I}^k \backslash \mathcal{I}^{k-1}, \quad \mathcal{I}^k = \{1,2,\cdots,m_k\}. \tag{4.13}$$

Then the set of the sparse grid interpolation points and the corresponding interpolation space are given as

$$\mathcal{X}_d^q = \bigcup_{d\le|\boldsymbol{i}|_1\le q}\Big((\mathcal{X}^{i_1}\backslash\mathcal{X}^{i_1-1})\times\cdots\times(\mathcal{X}^{i_1}\backslash\mathcal{X}^{i_1-1})\Big),\quad q\ge d, \tag{4.14}$$

$$V_d^q = \mathrm{span}\{\tilde{\phi}_{\boldsymbol{k}}(\boldsymbol{x}),\, \boldsymbol{k}\in\mathcal{I}_d^q\}\quad q\ge d, \tag{4.15}$$

where $\tilde{\phi}_{\boldsymbol{k}}$ can be chosen as the hierarchical interpolation basis defined in [40], or the Lagrange-type d-dimensional interpolation polynomial on points $\mathcal{X}_d^q$, which takes value 1 on $\boldsymbol{k}$-th interpolation point and 0 on the other points.

A commonly used 1-dimensional scheme is the Chebyshev-Gauss-Lobatto scheme, which uses the extrema of the Chebyshev polynomials as interpolation points:

$$x_j^i = -\cos\left(\frac{(j-1)\pi}{m_i-1}\right),\quad j=1,2,\cdots,m_i. \tag{4.16}$$

In order to obtain nested sets of points, m_i are chosen as

$$m_i = \begin{cases} 1, & i=1, \\ 2^{i-1}+1, & i>1, \end{cases} \tag{4.17}$$

with $x_1^1 := 0$. Define

$$F_d^k := \{f:[-1,1]^d\to\mathbb{R}\,|\,D^{\boldsymbol{\alpha}}f\in C([-1,1]^d),\ \forall\ |\boldsymbol{\alpha}|_\infty\le k\}. \tag{4.18}$$

Then for any function $f\in F_d^k$, with $\|f\|_{F_d^k} := \max_{|\boldsymbol{\alpha}|_\infty\le k}\|\partial^\alpha f\|_{L^\infty}\le 1$, the interpolation error on the above Chebyshev sparse grids are bounded as Theorem 8 in [36]:

$$\|f-\mathcal{A}(q,d)f\|_{L^\infty}\le c_{d,k}2^{-kq}q^{2d-1}\le c_{d,k}n^{-k}(\log n)^{(k+2)(d-1)+1}, \tag{4.19}$$

where $n=\mathrm{Card}(\mathcal{X}_d^q)=\mathrm{Card}(\mathcal{I}_d^q)=\mathcal{O}(2^q q^{d-1})$ is the number of points in the sparse grids, and $c_{d,k}$ is a constant that depends on d,k only. Note that if a different norm instead of the L^∞ norm is used, one can improve the result a little bit, but no results with error bound smaller than $\mathcal{O}(n^{-k})$ is known.

4.2 Error bounds of deep ReQU network approximation for multivariate functions with sparse structures

Now we discuss the ReQU network approximation of high-dimensional smooth functions with sparse polynomial expansions, which takes hyperbolic cross and sparse grid polynomial expansions as examples. We introduce the concept of *downward closed* polynomial space first. A linear polynomial space P_C is said to be downward closed if it satisfies the following: if d-dimensional polynomial $p(\boldsymbol{x})\in P_C$, then $\partial_{\boldsymbol{x}}^{\boldsymbol{k}}p(\boldsymbol{x})\in P_C$ for any $\boldsymbol{k}\in\mathbb{N}_0^d$, at the same time, there exists a set of bases that is composed of monomials only. It is easy to verify that the hyperbolic cross polynomial space X_N^d, the sparse grid polynomial interpolation space V_d^q, and the optimized hyperbolic cross space $X_{N,\gamma}^d$ are all downward closed. For a downward closed polynomial space, we have the following ReQU network representation results.

Theorem 4.1. *Let P_C be a downward closed linear space of d-dimensional polynomials with dimension n, then for any function $f \in P_C$, there exists a σ_2 neural network having no more than $\sum_{i=1}^d \lfloor \log_2 N_i \rfloor + d$ hidden layers, no more than $\mathcal{O}(n)$ activation functions and nonzero weights, can represent f exactly. Here N_i is the maximum polynomial degree with respect to the i-th coordinate.*

Proof. The proof is similar to Theorem 3.1. First, f can be written as a linear combination of monomials.

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{k}\in\chi_C} a_{\boldsymbol{k}} \boldsymbol{x}^{\boldsymbol{k}}, \tag{4.20}$$

where χ_C is the index set of P_C with cardinality n. Then we rearrange the summation as

$$f(\boldsymbol{x}) = \sum_{k_d=0}^{N_d} a_{k_d}^{x_1,\cdots,x_{k_{d-1}}} x_d^{k_d}, \quad a_{k_d}^{x_1,\cdots,x_{k_{d-1}}} := \sum_{(k_1,\cdots,k_{d-1})\in\chi_C^{k_d}} a_{k_1,\cdots,k_{d-1},k_d} x_1^{k_1}\cdots x_{d-1}^{k_{d-1}}, \tag{4.21}$$

where $\chi_C^{k_d}$ are $d-1$ dimensional downward closed index sets that depend on the index k_d. If each $a_{k_d}^{x_1,\cdots,x_{k_{d-1}}}$, $k_d = 0,1,\cdots,N_d$ can be exactly represented by a σ_2 network with no more than $\sum_{i=1}^{d-1} \lfloor \log_2 N_i \rfloor + (d-1)$ hidden layers, no more than $\mathcal{O}(\mathrm{Card}(\chi_C^{k_d}))$ nodes and nonzero weights, then $f(x)$ can be exactly represented by a σ_2 neural network with no more than $\sum_{i=1}^d \lfloor \log_2 N_i \rfloor + d$ hidden layers, no more than $\mathcal{O}(n)$ nodes and nonzero weights, since the operation $\sum_{k_d=0}^{N_d} a_{k_d}^{x_1,\cdots,x_{k_{d-1}}} x_d^{k_d}$ can be realized exactly by a σ_2 network with $\lfloor \log_2 N_d \rfloor + 1$ hidden layers and no more than $\mathcal{O}(N_d)$ nodes and nonzero weights. So, by mathematical induction, we only need to prove that when $d=1$ the theorem is satisfied, which is true by Theorem 2.2. □

Remark 4.1. According to Theorem 4.1, we have that:

1) For any $f \in X_N^d$, there exists a ReQU network with no more than $d\lfloor \log_2 N \rfloor + d$ hidden layers, no more than $\mathcal{O}(N(\log N)^{d-1})$ neurons and nonzero weights, that can represent f with no error.

2) For any $f \in X_{N,\gamma}^d$ with $0<\gamma<1$, there exists a ReQU network having no more than $d\lfloor \log_2 N \rfloor + d$ hidden layers, no more than $\mathcal{O}(N)$ neurons and nonzero weights, that can represent f with no error.

3) For any $f \in V_d^q$, there exists a ReQU network having no more than $d(q-d+2)$ hidden layers, no more than $\mathcal{O}(2^q q^{d-1})$ neurons and nonzero weights, that can represent f with no error.

Combining the results in Remarks 4.1 with (4.7), (4.10) and (4.19), we obtain the following theorem.

Theorem 4.2. *We have following results for ReQU network approximation of functions in* $\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, $\boldsymbol{\alpha},\boldsymbol{\beta}\in(-1,\infty)^d$, $m\geq 1$ *and* $F^k_d(I^d)$, $k\geq 1$*:*

1) *For any function* $u\in\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, $m\geq 1$ *with* $|u|_{\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}\leq 1/D_1$, *any* $\varepsilon>0$, *there exists a ReQU network* Φ^u_ε *with no more than* $\frac{d}{m}\log_2\frac{1}{\varepsilon}+d$ *hidden layers, no more than* $\mathcal{O}\big(\varepsilon^{-1/m}(\frac{1}{m}\log\frac{1}{\varepsilon})^{d-1}\big)$ *nodes and nonzero weights, such that*

$$\|R_{\sigma_2}(\Phi^u_\varepsilon)-u\|_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}\leq\varepsilon. \tag{4.22}$$

2) *For any function* $u\in\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, $m\geq 1$ *with* $|u|_{\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}\leq 1/D_2$, *any* $\varepsilon>0$, $0<\gamma<1$, *there exists a ReQU network* Φ^u_ε *with no more than* $\frac{d}{m(1-\gamma(1-\frac{1}{d}))}\log_2\frac{1}{\varepsilon}+d$ *hidden layers, no more than* $\mathcal{O}\big(\varepsilon^{-1/[m(1-\gamma(1-\frac{1}{d}))]}\big)$ *nodes and nonzero weights, such that*

$$\|R_{\sigma_2}(\Phi^u_\varepsilon)-u\|_{\omega^{\boldsymbol{\alpha},\boldsymbol{\beta}}}\leq\varepsilon. \tag{4.23}$$

3) *For any function* $f\in F^k_d(I^d)$, $k\geq 1$ *with* $\|f\|_{F^k_d}\leq 1$, *any* $\varepsilon>0$, *there exists a ReQU network* Ψ^f_ε *with no more than* $\mathcal{O}\big(\frac{d}{k}\log_2\frac{1}{\varepsilon}+d\big)$ *hidden layers, no more than* $\mathcal{O}\big(\varepsilon^{-\frac{1+\delta}{k}}(\frac{1+\delta}{k}\log_2\frac{1}{\varepsilon})^{d-1}\big)$ *nodes and nonzero weights, such that*

$$\|R_{\sigma_2}(\Psi^f_\varepsilon)-f\|_{L^\infty}\leq\varepsilon, \tag{4.24}$$

where $\delta>0$ *can be taken very close to* 0 *for small enough* ε.

Remark 4.2. Taking $m=2$ in Theorem 4.2, we obtain the following result: For any function $u\in\mathcal{K}^2_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, with $|u|_{\mathcal{K}^2_{\boldsymbol{\alpha},\boldsymbol{\beta}}}\leq 1/D_1$, and $\varepsilon>0$ there exists a ReQU network Φ^u_ε with no more than $\frac{d}{2}\log_2\frac{1}{\varepsilon}+d$ hidden layers, no more than $\mathcal{O}\big(\varepsilon^{-1/2}(\frac{1}{2}\log\frac{1}{\varepsilon})^{d-1}\big)$ nodes and nonzero weights, that approximates u with a tolerance ε. A result of using ReLU networks approximating similar functions is recently given by Montanelli and Du [50]. To approximate a function in $\mathcal{K}^2_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$ with tolerance ε, they constructed a ReLU network with $\mathcal{O}(|\log_2\varepsilon|\log_2 d)$ layers and $\mathcal{O}(\varepsilon^{-\frac{1}{2}}|\log_2\varepsilon|^{\frac{3}{2}(d-1)+1}\log_2 d)$ nonzero weights. Comparing the two results, we find that, while the number of layers required by ReQU networks might be larger than ReLU networks, the overall complexity of the ReQU network is $|\log_2\varepsilon|^d$ times smaller than that of ReLU network.

Remark 4.3. When one use optimized hyperbolic cross polynomial approximation for functions in $\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}(I^d)$, with $|u|_{\mathcal{K}^m_{\boldsymbol{\alpha},\boldsymbol{\beta}}}\leq 1/D_2$, the exponential growth on d with a base related to $1/\varepsilon$ in the required ReQU network size is removed. Thus, in this case it seems that the curse of dimensionality does not exist any more. But we note that, the constant D_2 and the implicit constant hidden in the big $\mathcal{O}$ notation, still depend on d. In practice, the error bound given by the second case may not be better than the first case.

5 Some preliminary numerical results

In this section, we present some numerical results to verify that the construction algorithms proposed are numerically stable and efficient. We first present the results of representing univariate monomials in Table 1. The maximum norm error in this table is calculated by taking the maximum difference on 100 randomly choose points in $[-1,1]$. The results show that the ReQU network we constructed can achieve machine accuracy, which means our approach is numerically stable.

Table 1: Representation of monomials x^n.

Degree n	L	#weight	#node	L^∞-Error
3	3	38	10	4.44e-16
7	4	64	15	2.22e-16
15	5	89	20	9.99e-16
31	6	114	25	7.77e-16
63	7	139	30	6.11e-16
127	8	164	35	2.22e-16

Similar results for representing univariate polynomials are given in Table 2. Here, the coefficients of the power series are generated randomly according to standard normal distribution. These results also verify our approach is stable and efficient.

Table 2: Representation of univariate polynomials of degree n.

Degree n	L	#Weight	#Node	L^∞-Error
3	3	66	14	1.78e-15
7	4	188	31	1.78e-15
15	5	429	64	4.44e-15
31	6	910	129	5.33e-15
63	7	1871	258	5.33e-15
127	8	3792	515	5.33e-15

Numerical tests for 2-dimensional polynomials in tensor-product space and hyperbolic cross space are presented in Tables 3 and 4, respectively. The coefficients of corresponding power series are all randomly generated according to standard normal distribution. The results verify the stability and efficiency of our method.

Next, we present some results of approximated 1-dimensional and 2-dimensional smooth functions using our approach, and compare them with trained ReLU network approximations. We first show the results of approximating $\sin(x)$ using ReQU network of our approach and ReLU network with randomly initialized coefficients. The ReQU network is constructed using proposed method based on a polynomial approximation of degree 8 and then trained by gradient descent method. The result is shown in the left

Table 3: Representation of polynomials in tensor-product space Q_N^2.

Degree N	L	#Weight	#Node	L^∞-Error
3	5	378	64	1.11e-15
7	7	1570	246	8.88e-15
15	9	6376	988	1.60e-14
31	11	25758	4002	7.11e-14
63	13	103668	16168	8.88e-14

Table 4: Representation of polynomials in hyperbolic cross polynomial space.

Degree N	L	#Weight	#Node	L^∞-Error
7	7	1254	217	3.55e-15
15	9	3277	554	1.24e-14
31	11	8022	1351	5.32e-14
63	13	19039	3196	2.24e-14
127	15	44052	7393	4.26e-14

plot of Fig. 3. For the ReLU network approximation, we take 5 ReLU networks with same structure (8 layers of hidden nodes with each layer has 64 ReLU nodes, full connected) are trained using mini-batch stochastic gradient descent method. The best result among the 5 ReLU networks is shown in the right plot of Fig. 3. Note that the number of hidden nodes used by the ReQU network is less than 64, and it give much better results than the trained ReLU network. By training the constructed ReQU network, the approximation error can be further reduced. Similar results for approximating 2-dimensional function $\sin(x)\sin(y)$ are presented in Fig. 4.

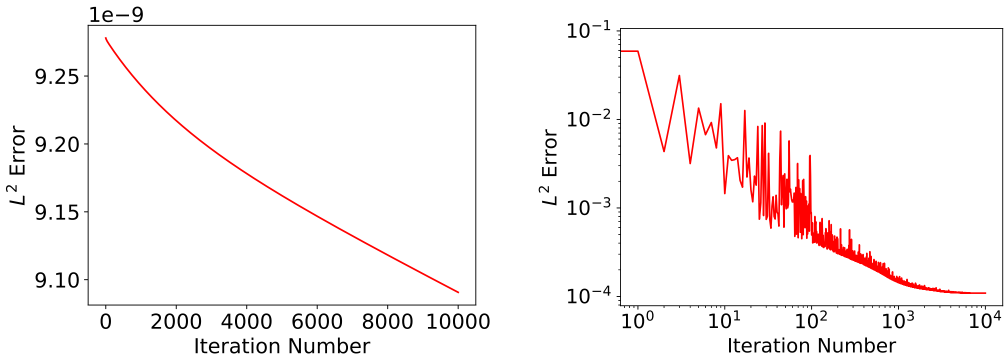

Figure 3: Approximating $\sin(x)$ function using ReQU and ReLU neural networks. Left: result of ReQU network initialized by polynomials of degree 8 and then trained by a gradient descent method. Right: result of ReLU network (8 fully connected hidden layers with each one has 64 ReLU nodes) with a random initialization and trained by a mini-batch gradient descent method.

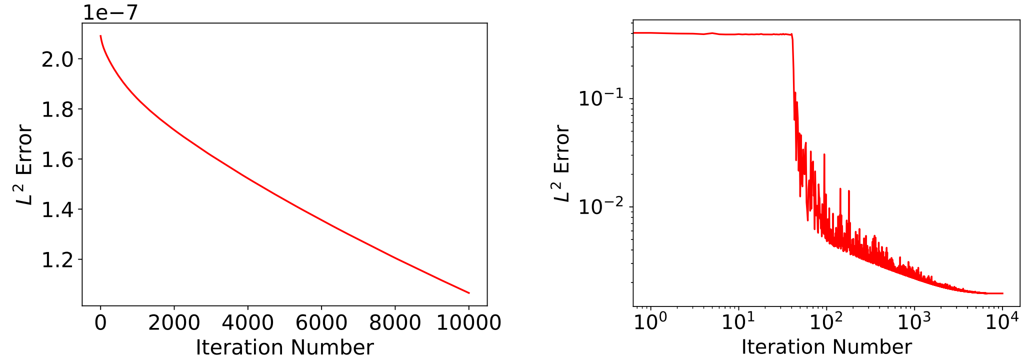

Figure 4: Approximating $\sin(x)\sin(y)$ using ReQU and ReLU neural networks. Left: result of ReQU network initialized by a 2-d polynomial in tensor-product space Q_N^2 $(N=9)$ and then trained by a gradient descent method; Right: result of ReLU network (8 fully connected hidden layers, each one has 128 ReLU nodes) with a random initialization and then trained by a mini-batch gradient descent method.

6 Conclusion and future work

In this paper, we gave constructive proofs of some error bounds for approximating smooth functions by deep neural networks using RePU function as the activation functions. The proofs rely on the fact that polynomials can be represented by RePU networks with no approximation error. We construct several optimal algorithms for such representations, in which polynomials of degree no more than n are converted into a ReQU network with $\mathcal{O}(\log_2 n)$ layers, and the size of the network is of the same scale as the dimension of the polynomial space to be approximated. Then by using the classical polynomial approximation theory, we obtain upper error bounds for ReQU networks approximating smooth functions, which show clear advantages of using ReQU activation function, comparing to the existing results for ReLU networks. In general, the ReLU network required to approximate a sufficient smooth function, is $\mathcal{O}(\log\frac{1}{\varepsilon})$ times larger than the corresponding ReQU network. Here ε is the approximation error. To achieve ε-approximation for $f\in B^{\infty}_{\alpha,\beta}$, the number of layer of ReQU network required to obtain this approximation is $\mathcal{O}(\log_2\log\frac{1}{\varepsilon})$, while the corresponding best known results is $\mathcal{O}(\log\frac{1}{\varepsilon})$ for ReLU network. For high dimensional functions with bounded mixed derivatives, we give error bounds that have a weaker exponentially dependence on d, by using hyperbolic cross/sparse grid spectral approximation, in particular if optimized hyperbolic cross polynomial projections are used, there is no term related to ε is exponentially dependent on d. Since only global polynomial approximations are considered in this paper, the results obtained also hold for deep networks with non-rectified power units. The use of rectified units gives the neural network the ability to approximate piecewise smooth functions efficiently, which will be analyzed in a separate paper.

Our constructions of RePU network also reveal the close relation between the depth of the RePU network and the "order" of polynomial approximation. The advantage of using *deep* over *shallow* neural ReQU networks is clearly shown by our constructive proofs: by using one hidden layer, a ReQU network can only represent piecewise quadratic polynomials; by using n hidden layers, a ReQU network can represent piecewise polynomials of degree up to $\mathcal{O}(2^n)$. The ReQU networks we built for approximating smooth functions all have a tree-like structure, and are sparsely connected. This may give some hints on how to design appropriate structures of neural networks for some practical applications.

We have shown theoretically that for approximating sufficient smooth functions, ReQU networks are superior to ReLU networks in terms of approximation error. We also present efficient and stable algorithm to construct ReQU network based on polynomial approximation. Our preliminary results demonstrated that our constructions are numerically stable and efficient. The constructed neural network can be regarded as a good initial of RePU network and further trained to get better results. For low dimensional problems, this approach is much more accurate than the results obtained by direct training a randomly initialized ReLU neural networks.

In practical applications, the functions to be approximated may have different kinds of non-smoothness, which are problem dependent. The training method is another key factor that affects the application of neural networks. We will continue our study in these directions. In particular, we will study the approximation error of piecewise smooth functions with deep ReQU networks, and investigate whether those popular training methods proposed to train ReLU networks are efficient for training RePU networks. Meanwhile, we will try deep RePU networks on some practical problems where the underlying functions are smooth, e.g. minimum action methods for large PDE systems [61], PDEs with random coefficients [62], and moment closure problem in complex fluid [63] and turbulence modeling [64], etc.

Acknowledgments

We are indebted to Prof. Jie Shen and Prof. Li-Lian Wang for their stimulating conversations on spectral methods. We would like also to think Prof. Christoph Schwab and Prof. Hrushikesh N. Mhaskar for providing us some related references. This work was partially supported by China National Program on Key Basic Research Project 2015CB856003, NNSFC Grant 11771439, 91852116, and China Science Challenge Project, no. TZ2018001. The computations were performed on the PC clusters of State Key Laboratory of Scientific and Engineering Computing of Chinese Academy of Sciences.

References

[1] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[2] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems*, pages 153–160, 2007.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Process. Mag.*, 29, 2012.

[6] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[7] J. Han, L. Zhang, R. Car, and W. E. Deep potential: A general representation of a many-body potential energy surface. *Communications in Computational Physics*, 23(3), 2018.

[8] J. Han, A. Jentzen, and W. E. Solving high-dimensional partial differential equations using deep learning. *PNAS*, 115(34):8505–8510, 2018.

[9] L. Zhang, J. Han, H. Wang, R. Car, and W. E. Deep potential molecular dynamics: A scalable model with the accuracy of quantum mechanics. *Phys. Rev. Lett.*, 120(14):143001, 2018.

[10] C. M. Strofer, J.-L. Wu, H. Xiao, and E. Paterson. Data-driven, physics-based feature extraction from fluid flow fields using convolutional neural networks. *Communications in Computational Physics*, 25(3), 2019.

[11] C. Ma, J. Wang, and W. E. Model reduction with memory and the machine learning of dynamical systems. *Communications in Computational Physics*, 25(4), 2019.

[12] M. Cavaglia, K. Staats, and T. Gill. Finding the origin of noise transients in LIGO data with machine learning. *Communications in Computational Physics*, 25(4), 2019.

[13] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems*, 2(4):303–314, 1989.

[14] H. N. Mhaskar. Neural networks for optimal approximation of smooth and analytic functions. *Neural Computation*, 8(1):164–177, 1996.

[15] O. Delalleau and Y. Bengio. Shallow vs. deep sum-product networks. In *NIPS*, page 9, 2011.

[16] M. Telgarsky. Representation benefits of deep feedforward networks. *ArXiv150908101 Cs*, 2015.

[17] R. Eldan and O. Shamir. The power of depth for feedforward neural networks. *JMLR Workshop Conf. Proc.*, 49:1–34, 2016.

[18] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the 14 Th International Conference on Artificial Intelligence and Statistics*, volume 15, pages 315–323, Fort Lauderdal, 2011. JMLR.

[19] S. Liang and R. Srikant. Why deep neural networks for function approximation? *ArXiv161004161 Cs*, 2016.

[20] M. Telgarsky. Benefits of depth in neural networks. In *JMLR: Workshop and Conference Proceedings*, volume 49, pages 1–23, 2016.

[21] D. Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural Netw.*, 94:103–114, 2017.

[22] P. Petersen and F. Voigtlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *Neural Netw.*, 108:296–330, 2018.

[23] W. E and Q. Wang. Exponential convergence of the deep neural network approximation for analytic functions. *Sci. China Math.*, 61(10):1733–1740, 2018.

[24] J. He, L. Li, J. Xu, and C. Zheng. ReLU deep neural networks and linear finite elements. *ArXiv180703973 Math*, 2018.

[25] J. A. A. Opschoor, P. C. Petersen, and Ch. Schwab. Deep ReLU networks and high-order finite element methods. Technical Report 7, SAM ETH Zurich, 2019.

[26] H. N. Mhaskar. Approximation properties of a multilayered feedforward artificial neural network. *Adv Comput Math*, 1(1):61–80, 1993.

[27] C. K. Chui, X. Li, and H. N. Mhaskar. Neural networks for localized approximation. *Math. Comp.*, 63(208):607–623, 1994.

[28] C. K. Chui and H. N. Mhaskar. Deep nets for local manifold learning. *Front. Appl. Math. Stat.*, 4, 2018.

[29] J. A. A. Opschoor, Ch. Schwab, and J. Zech. Exponential ReLU DNN expression of holomorphic maps in high dimension. Technical Report 35, SAM ETH Zurich, 2019.

[30] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl Akad Nauk SSSR*, 148(5):1042–1045, 1963.

[31] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numer.*, 13:1–123, 2004.

[32] M. Griebel and J. Hamaekers. Sparse grids for the Schrödinger equation. *Math. Model. Numer. Anal.*, 41(2):215–247, 2007.

[33] J. Shen and L.-L. Wang. Sparse spectral approximations of high-dimensional problems based on hyperbolic cross. *SIAM J Numer Anal*, 48(4):1087–1109, 2010.

[34] D. Dũng, V. Temlyakov, and T. Ullrich. *Hyperbolic Cross Approximation*. Advanced Courses in Mathematics. CRM Barcelona. Birkhäuser/Springer, Cham, 2018.

[35] T. Gerstner and M. Griebel. Numerical integration using sparse grids. *Numer. Algorithms*, 18(3):209–232, 1998.

[36] V. Barthelmann, E. Novak, and K. Ritter. High dimensional polynomial interpolation on sparse grids. *Adv. Comput. Math.*, 12(4):273–288, 2000.

[37] J. Shen, L.-L. Wang, and H. Yu. Approximations by orthonormal mapped Chebyshev functions for higher-dimensional problems in unbounded domains. *J. Comput. Appl. Mathemaitcs*, 265:264–275, 2014.

[38] H. J. Bungartz. An adaptive Poisson solver using hierarchical bases and sparse grids. In *Iterative Methods in Linear Algebra*, pages 293–310, Brussels, Belgium, 1992. Amsterdam: North-Holland.

[39] Q. Lin, N. Yan, and A. Zhou. A sparse finite element method with high accuracy: Part I. *Numer. Math.*, 88(4):731–742, 2001.

[40] J. Shen and H. Yu. Efficient spectral sparse grid methods and applications to high-dimensional elliptic problems. *SIAM J. Sci. Comput.*, 32(6):3228–3250, 2010.

[41] J. Shen and H. Yu. Efficient spectral sparse grid methods and applications to high-dimensional elliptic equations II: Unbounded domains. *SIAM J. Sci. Comput.*, 34(2):1141–1164, 2012.

[42] Z. Wang, Q. Tang, W. Guo, and Y. Cheng. Sparse grid discontinuous Galerkin methods for high-dimensional elliptic equations. *J. Comput. Phys.*, 314:244–263, 2016.

[43] Z. Rong, J. Shen, and H. Yu. A nodal sparse grid spectral element method for multi-dimensional elliptic partial differential equations. *Int. J. Numer. Anal. Model.*, 14(4-5):762–783, 2017.

[44] H. Yserentant. The hyperbolic cross space approximation of electronic wavefunctions. *Numer. Math.*, 105(4):659–690, 2007.

[45] G. Avila and T. Carrington. Solving the Schroedinger equation using Smolyak interpolants. *J. Chem. Phys.*, 139(13):134114, 2013.

[46] J. Shen, Y. Wang, and H. Yu. Efficient spectral-element methods for the electronic Schrödinger equation. In J. Garcke and D. Pflüger, editors, *Sparse Grids and Applications –*

Stuttgart 2014, Lecture Notes in Computational Science and Engineering, pages 265–289. Springer International Publishing, 2016.

[47] Ch. Schwab and R. A. Todor. Sparse finite elements for stochastic elliptic problems – higher order moments. *Computing*, 71(1):43–63, 2003.

[48] F. Nobile, R. Tempone, and C. Webster. A sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM J. Numer. Anal.*, 46(5):2309–2345, 2008.

[49] F. Nobile, L. Tamellini, F. Tesei, and R. Tempone. An adaptive sparse grid algorithm for elliptic PDEs with lognormal diffusion coefficient. In J. Garcke and D. Pflüger, editors, *Sparse Grids and Applications – Stuttgart 2014*, volume 109, pages 191–220. Springer International Publishing, Cham, 2016.

[50] H. Montanelli and Q. Du. New error bounds for deep ReLU networks using sparse grids. *SIAM J. Math. Data Sci.*, 1(1):78–92, 2019.

[51] P. Petrushev. Approximation by ridge functions and neural networks. *SIAM J. Math. Anal.*, 30(1):155–189, 1998.

[52] W. E and B. Yu. The deep Ritz method: A deep learning-based numerical algorithm for solving variational problems. *Commun. Math. Stat.*, 6(1):1–12, 2018.

[53] W. Gautschi. Optimally scaled and optimally conditioned vandermonde and vandermonde-like matrices. *BIT Numerical Mathematics*, 51(1):103–125, 2011.

[54] J. Shen, T. Tang, and L.-L. Wang. *Spectral Methods : Algorithms, Analysis and Applications*. Springer, 2011.

[55] B. Li, S. Tang, and H. Yu. PowerNet: Efficient representations of polynomials and smooth functions by deep neural networks with rectified power units. *arXiv:1909.05136*, 2019.

[56] R. A. DeVore, R. Howard, and C. Micchelli. Optimal nonlinear approximation. *Manuscripta Math*, 63(4):469–478, 1989.

[57] F. Voigtlaender and P. Petersen. Approximation in $L^p(\mu)$ with deep ReLU neural networks. *ArXiv190404789 Cs Math*, 2019.

[58] D. Yarotsky. Optimal approximation of continuous functions by very deep ReLU networks. In *Conference on Learning Theory*, pages 639–649, 2018.

[59] I. H. Sloan and H. Wozniakowski. When are quasi-Monte Carlo algorithms efficient for high dimensional integrals? *J. Complex.*, 14(1):1–33, 1998.

[60] X. Wang and I. Sloan. Why are high-dimensional finance problems often of low effective dimension? *SIAM J. Sci. Comput.*, 27(1):159–183, 2005.

[61] X. Wan and H. Yu. A dynamic-solver-consistent minimum action method: With an application to 2D Navier-Stokes equations. *Journal of Computational Physics*, 331:209–226, 2017.

[62] E. Musharbash, F. Nobile, and T. Zhou. Error analysis of the dynamically orthogonal approximation of time dependent random PDEs. *SIAM J. Sci. Comput.*, 37(2):A776–A810, 2015.

[63] H. Yu, G. Ji, and P. Zhang. A nonhomogeneous kinetic model of liquid crystal polymers and its thermodynamic closure approximation. *Commun. Comput. Phys.*, 7(2):383, 2010.

[64] G. L. Mellor and T. Yamada. Development of a turbulence closure model for geophysical fluid problems. *Rev. Geophys.*, 20(4):851–875, 1982.