Ladder Polynomial Neural Networks

A thesis submitted by

Ruiyuan Gu

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Tufts University

May 2022

advisor: Liping Liu

Abstract

A combination of neural networks and polynomial functions offers some favorable theoretical properties to neural network. Polynomial neural networks limit the model functions to polynomials. However, it is hard to control the polynomial degree of these models. In this work, we devise the product activation function and then create the Ladder Polynomial Neural Network (LPNN). LPNN has a feedforward structure and can be trained as other neural networks. It has a polynomial model function that can take any polynomial degree. We show that this model is related to several previous polynomial models and has many advantages in the optimization process and Bayesian learning. In empirical study, deep LPNN models outperform other polynomial models in a series of regression and classification tasks. It also narrows the gap between polynomial models and well-developed feedforward neural network models.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Professor Liping Liu who gave me a golden opportunity to do this wonderful project. His guidance and patience makes this project possible. He also helps me a lot in both my research and my life.

I would also like to thank Professor Xiaozhe Hu who provided important instructions in this project.

Then, I would like to thank my committee for valuable feedback to this thesis.

Finally, I want to thank my friends and my family who always support me.

Contents

List of Tables

List of Figures

1 Introduction

In the last decade, new machine learning models based on the deep neural network have deeply changed many areas of the modern world. They have higher performance on multiple tasks than previous methods and motivate great quantities of successful commercial applications.

The complexity of those models guaranteed the high capacity, which means they can approximate complicated functions to mimic the actual relation between input and output. But at the same time, it is problematic to analyze those complex models or explain their behavior[11]. One reason for the low interpretability is the intricate model function. Even a small feedforward artificial neural network model with a few layers will have a nested model function, which is hard to understand the relationship between each input value and each output value. Those model functions lack favorable theoretical property and are mostly underresearched.

On the other side, we have many types of functions with good properties. One promising candidate of those functions is the polynomial function. As one of the most thoroughly researched types of functions, the polynomial function has many known good properties. A model with a polynomial model function will benefit from those properties. And multiple math tools can be applied to them. More important, polynomial functions have clear form and are easier for human to understand the relationship between input and output, which means a higher interpretability.

A model with polynomial function as model function is a polynomial model. One kind of methods is to construct such a model is representing polynomial coefficients in a polynomial function with some type of decomposition. To do the decomposition, previous works defined a polynomial kernel over input features and network parameters [3][2]. Some other works used a tensor-train decomposition as the coefficients of a polynomial model[5]. But those models are not neural network models.

As the most popular and capable model architecture in machine learning, it is more valuable to construct a neural network model with polynomial model function. Such a neural network model is

called a polynomial neural network (PNN). It bridges analysis of the general neural networks to the properties of polynomial functions[12]. Those polynomial models are polynomial-time learnable, and they can approximate other feedforward neural networks.

One method of constructing polynomial models is to use a specialized activation function. For example, a feedforward network with a quadratic activation function is a PNN, a polynomial model [12]. But it has a clear drawback: the polynomial order grows exponentially with its number of layers. So it cannot have an arbitrary order and its performance is not stable when the model is deep.

Based on the previous researches, we found that the activation function is the key component to modify the formula of model function. In my master thesis, as we want a polynomial neural network model with precise degree control and better performance, we devise a new activation function, product activation. We use it in a feedforward network to construct a new type of polynomial neural network, Ladder Polynomial Neural Networks (LPNNs).

In this new activation function, the hidden layer is multiplied to a linear transformation of the input feature. As a feedforward function, we can train the LPNNs with many training techniques like batch normalization[8] and dropout[18]. For each layer with product activation, the polynomial order of the model will increase by 1, so the order can be precisely controlled. And it can also cover two previous decomposition models as special cases. At the same time, this model has a useful property in Bayesian learning: the moments of a LPNN's outputs can be computed in closed-form when network weights are stochastic.

We tested the LPNN model with multiple classifications and regression tasks on a list of datasets. The experiment result shows our model outperforms previous polynomial models. We also found that popular technical works for neural networks can also improve LPNN'S performance. And our result also shows that Gaussian distributions well approximate a LPNN's network output when the network is given a Gaussian prior.

2 Related Work

2.1 Feedforward network with quadratic activations

We first define the general form of a feedforward neural network. Suppose the input to the neural network is a feature vector $\mathbf{x} \in \mathbb{R}^{d_0}$ and denote $\mathbf{h}^0 = \mathbf{x}$. Suppose the network has L hidden layers, with each layer $\ell \in \{1, \ldots, L\}$ takes the input $\mathbf{h}^{\ell-1}$ and has the output $\mathbf{h}^\ell$. Note that superscripts of hidden vectors and weight matrices always denote layer indices, not exponents.

Then Each layer in the model is defined by

$$\mathbf{h}^\ell = \sigma\left(\mathbf{W}^\ell \mathbf{h}^{\ell-1}\right). \tag{1}$$

Here $\mathbf{W}^\ell$ is the weight matrix for layer ℓ, and $\sigma(\cdot)$ is the activation function.

Livni et al. [12] proposed quadratic activation function. The definition of quadratic activation functions:

$$\sigma_{quad}(\mathbf{W}^\ell \mathbf{h}^{\ell-1}) = \left(\mathbf{W}^\ell \mathbf{h}^{\ell-1}\right)^2. \tag{2}$$

Livni et al. also analyze the properties of feedforward networks with quadratic activations (FF-QUADS). They show that FF-QUADS is as expressive as networks with threshold activation, and they can be learned in polynomial time. Kileel et al. analyze the algebraic structure of polynomial functions behind FF-QUADS[9]. Du and Lee show that training a one-hidden-layer FF-QUAD is efficient when the model is overly parameterized[6]. There are also other works proposed special optimization methods for shadow FF-QUAD models. Lin et al. consider a the second order linear model which is a special form of one layer FF-QUADS[10]. Soltani et al. use low-Rank matrix estimation to improve

the training process of FF-QUADS with one hidden layer[16, 17].

2.2 Decomposition models

Blondel et al. construct polynomial models with polynomial kernels [3]. They also show that factorization machines[15] can be constructed in the same way with ANOVA kernels. In other works, they propose high order factorization machines with high order ANOVA kernels[1], and extend factorization machines and polynomial networks to output multiple values[2]. Chen et al. use tensor-train decomposition [14] to express the coefficients of a polynomial model[5].

3 The Polynomial Neural Network

Similar to the quadratic activation, suppose the input to the neural network is a feature vector $\mathbf{x} \in \mathbb{R}^{d_0}$, and denote $\mathbf{h}^0 = \mathbf{x}$. Suppose the network has L hidden layers, with each layer $\ell \in \{1, \ldots, L\}$ takes the input $\mathbf{h}^{\ell-1}$ and has the output $\mathbf{h}^\ell$. Each layer is defined by

$$\mathbf{h}^\ell = \sigma\left(\mathbf{W}^\ell \mathbf{h}^{\ell-1}\right). \tag{3}$$

Still, $\sigma(\cdot)$ is the activation function and $\mathbf{W}^\ell$ is the weight matrix for layer ℓ. Then we have $\mathbf{W}^\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ when $\mathbf{h}^{\ell-1}$ has $d_{\ell-1}$ entries and $\mathbf{h}^\ell$ has d_ℓ entries.

Then we propose a new activation, the *product activation* $\sigma_p(\cdot)$ in the neural network.

The definition of this activation function is

$$\sigma_p(\mathbf{u}; \mathbf{V}, \mathbf{x}) = \mathbf{u} \odot (\mathbf{V}\mathbf{x}). \tag{4}$$

Here $\odot$ is the element-wise product or the Hadamard product. The learnable parameter $\mathbf{V}$ is a matrix

and $\mathbf{V} \in \mathbb{R}^{d \times d_0}$ when $\mathbf{u}$ has d entries.

Since $\mathbf{u} = \mathbf{W}^\ell \mathbf{h}^{\ell-1}$ is a function of $\mathbf{x}$, the activation is nonlinear in $\mathbf{u}$. Particularly, if $\mathbf{u}$ is a polynomial function of $\mathbf{x}$, then $\sigma_p(\mathbf{u}; \mathbf{V}, \mathbf{x})$ is also a polynomial function of $\mathbf{x}$ with the polynomial order increased by 1.

The product activation is inspired by the attention layer in a Transformer [19]. In transformer architecture, multiplication between query, key and value vectors is necessary to calculate the attention score. And these three vectors are obtained by multiplying input feature vector with a learnable matrix respectively. Self-attention are mainly used in natural language processing and computer vision, and has a special explanation. But the effectiveness of self-attention shows that the multiplication between features could be a promising way to processing the information. The product activation applies this idea and remove all other non-linear functions in the self-attention calculation. The product activation is unique from all previous activation functions which cannot output a product between two features.

Then we use product activations in a feedforward structure to construct a LPNN. We use a different matrix $\mathbf{V}^\ell$ for the product activation in each layer ℓ. Suppose $\mathbf{h}_L$ is the output of the neural network, the function of the LPNN is formally defined as $\text{LPNN}(\mathbf{x}; \theta) = \mathbf{h}^L$,

$$\mathbf{h}^0 = \mathbf{x}, \tag{5}$$

$$\mathbf{h}^\ell = \sigma_p(\mathbf{W}^\ell \mathbf{h}^{\ell-1}; \mathbf{V}^\ell, \mathbf{x}) = \mathbf{W}^\ell \mathbf{h}^{\ell-1} \odot (\mathbf{V}^\ell \mathbf{x}), \quad \ell = 1, 2, \ldots, L. \tag{6}$$

Let $\theta = \left(\mathbf{W}^1, \ldots, \mathbf{W}^L, \mathbf{V}^1, \ldots, \mathbf{V}^L\right)$ denote all network parameters.

Then consider the order of polynomial function provided by the LPNN model. For the first hidden layer $\mathbf{h}^1$, we have $\mathbf{h}^1 = (\mathbf{W}^1 \mathbf{x}) \odot (\mathbf{V}^1 \mathbf{x})$. Let the first hidden layer has d_1 hidden units, we have $^1 \in \mathbb{R}^{d_1}$ $\mathbf{W}^1, \mathbf{V}^1 \in \mathbb{R}^{d_1 \times d_0}$. Without loss of generality, we can assume that the model have more than 1 input features $\mathbf{x} = [x_1, x_2, ..., x_{d_0}]$.

To figure out the form of elements in $\mathbf{h}^1$, we need to expand them. Consider the first element in $\mathbf{h}^1$: h_1^1. It is the product of two degree-1 multivariate polynomials $\mathbf{W}_1^1\mathbf{x}$ and $\mathbf{V}_1^1\mathbf{x}$. Let $\mathbf{M}_i^1$ means the i-th row of $\mathbf{M}^1$ and $M_{i,j}^1$ means the element at i-th row and j-th column.

Then we have

$$\mathbf{W}_1^1\mathbf{x} = W_{1,1}^1 x_1 + W_{1,2}^1 x_2 + ... + W_{1,d_0}^1 x_{x_0}, \tag{7}$$

$$\mathbf{V}_1^1\mathbf{x} = V_{1,1}^1 x_1 + V_{1,2}^1 x_2 + ... + V_{1,d_0}^1 x_{x_0}, \tag{8}$$

$$h_1^1 = \mathbf{W}_1^1\mathbf{x} \cdot \mathbf{V}_1^1\mathbf{x} = \sum_{i=1}^{d_0}\sum_{j=1}^{d_0}(W_{1,i}^1 V_{1,j}^1 + W_{1,j}^1 V_{1,i}^1)x_i x_j \tag{9}$$

Obviously, elements in the vector $\mathbf{W}^1\mathbf{x}$ and $\mathbf{V}^1\mathbf{x}$ are order-1 multivariate polynomials of input features, and elements in $\mathbf{h}^1$ has a order of 2. In fact, each product activation increase the order by 1, so the hidden layer $\mathbf{h}^\ell$ is an order $(\ell + 1)$ polynomial. The entire network is an polynomial function of the input with order $L + 1$.

We further re-write the function with simple additions and multiplications. The i-th entry of $\mathbf{h}^\ell$ is

$$h_i^\ell = \left(\mathbf{W}_i^\ell \mathbf{h}^\ell\right)\left(\mathbf{V}_i^\ell \mathbf{x}\right). \tag{10}$$

Here $\mathbf{W}_i^\ell$ and $\mathbf{V}_i^\ell$ are the i-th rows of $\mathbf{W}$ and $\mathbf{V}$ respectively, so both $\left(\mathbf{W}_i^\ell \mathbf{h}^\ell\right)$ and $\left(\mathbf{V}_i^\ell \mathbf{x}\right)$ are scalars.

After expanding all product activations and taking out all summations, the i_L-th entry of the network function $\mathrm{LPNN}(\mathbf{x}; \theta)$ is

$$h_{i_L}^L = \sum_{i_{(L-1)}=1}^{d_{(L-1)}} \cdots \sum_{i_1=1}^{d_1}\left[\left(\prod_{\ell=2}^{L}\mathbf{W}_{i_\ell, i_{\ell-1}}^\ell\right)\left(\mathbf{W}_{i_1}^1\mathbf{x}\right)\left(\prod_{\ell=1}^{L}\mathbf{V}_{i_\ell}^\ell\mathbf{x}\right)\right]. \tag{11}$$

This equation further show that the polynomial order of the network is $L + 1$.

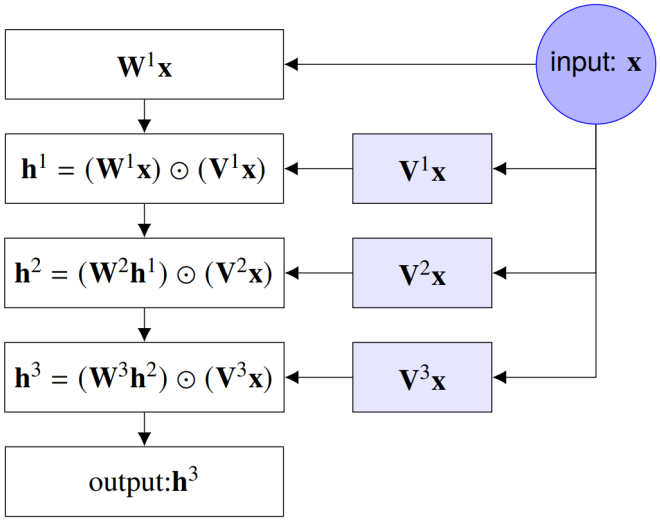


Figure 1: The Structure of a LPNN with 3 hidden layers

Then we noticed a problem in this framework: the polynomial can only contains terms with same degree, as the previous example shows. This will definitely limit the possible polynomials can be covered and reduce the complexity or capacity of the model. Similar to the polynomial regression, it is necessary to include terms with all lower orders to improve the performance of LPNN models. To achieve this, we need to include intercept vectors. One way to achieve this is adding constant term to the input and each layer.

$$\mathbf{h}^\ell = \sigma_p(\mathbf{W}^\ell \mathbf{h}^{\ell-1} + \mathbf{b}_w^\ell; \mathbf{V}^\ell, \mathbf{x} + \mathbf{b}_x^\ell). \tag{12}$$

In the implementation, it will be easier to rewrite the input as following form,

$$\hat{\mathbf{h}}^\ell = \begin{bmatrix} \mathbf{h} \\ 1 \end{bmatrix}, \quad \hat{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \tag{13}$$

And add an additional column to $\mathbf{W}^\ell$ and $\mathbf{V}^\ell$: $\mathbf{W}^\ell \in \mathbb{R}^{d_\ell \times (d_{\ell-1}+1)}$, $\mathbf{V}^\ell \in \mathbb{R}^{d_\ell \times (d_0+1)}$ when $\mathbf{h}^{\ell-1}$ has $d_{\ell-1}$ entries and $\mathbf{h}^\ell$ has d_ℓ entries. The intercept vectors $\mathbf{b}_w^\ell$ and $\mathbf{x} + \mathbf{b}_x^\ell$ will be integrated into those weight matrices.

then we still have the previous form, $\hat{\mathbf{h}}^\ell = \sigma_p(\mathbf{W}^\ell \hat{\mathbf{h}}^{\ell-1}; \mathbf{V}^\ell, \hat{\mathbf{x}})$, and then all previous derivations still apply. For notational simplicity, the following analysis continues to use notations without the intercept term.

4 Analysis

In this section, we perform some analysis to LPNN. The first part is a discussion about the connection between LPNN and a few previous decomposition models, which shows that LPNN covers those decomposition models. In the next part, we highlight a few properties of LPNN, demonstrating why polynomial function is beneficial. Then we verify that LPNN can be trained with popular training methods like batch normalization and dropout. Finally we analyze LPNN from the perspective of neural networks and show its properties in Bayesian learning.

4.1 Relation with decomposition models

Relation with polynomial kernels. We first show that the polynomial networks constructed from polynomial kernel functions by Blondel et al. [3] are special cases of the LPNN model.

Theorem 4.1. *The learning models in the form of* $y(\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\lambda + \mathbf{p}_k^\top \mathbf{x})^m$ *[3] can be written as a LPNN function.*

Proof. The polynomial kernel function, $\mathcal{P}^m(\mathbf{p}, \mathbf{x}) = (\lambda + \mathbf{p}^\top \mathbf{x})^m, \mathbf{p}, \mathbf{x} \in \mathcal{R}^d$ is an exponential function. For a single polynomial kernel function with only one output value, we may reduce the hidden unit in each layer of LPNN to 1 to simplify the problem. $\mathcal{P}^m(\mathbf{p}, \mathbf{x})$ is a m-order polynomial, which means the LPNN should have $m - 1$ layers. In LPNN, it is straightforward to construct a special LPNN layer in a form of an exponential function:

$$\mathbf{h}^\ell = \sigma_p((\lambda + \mathbf{p}^\top \mathbf{x})^\ell; \lambda + \mathbf{p}^\top \mathbf{x}) = (\lambda + \mathbf{p}^\top \mathbf{x})^\ell \cdot (\lambda + \mathbf{p}^\top \mathbf{x}) = (\lambda + \mathbf{p}^\top \mathbf{x})^{\ell+1}.$$

As result, an exponential function can be implemented by a LPNN model.

A polynomial kernel function can be written as a LPNN function in this method: we append an element 1 to the feature vector $\mathbf{x}$ as the new input $[\mathbf{x}, 1]$ to the network. Set $\mathbf{W}^1 = [\mathbf{p}^\top, \lambda]$, $\mathbf{V}^\ell = [\mathbf{p}^\top, \lambda]$ for all $\ell = 1, \ldots, m-1$. They are all matrices with only one row. Set $\mathbf{W}^\ell$ to be a single element matrix $[1]$ for $\ell = 2, \ldots, L$, then $\text{LPNN}(\mathbf{x}; \theta)$ is equivalent to the kernel by (11).

To get a weighted sum of multiple kernels, we can just put each kernel in each row of $\mathbf{W}^1$ and every $\mathbf{V}^\ell$. Then we set all $\mathbf{W}^\ell, \ell = 2, \ldots, L$ to the identity matrix with size K. Then all these kernels work in parallel in the LPNN network. Finally, we add an extra layer to take the weighted sum of the values of these kernels. □

This proof indicates that the model constructed from polynomial kernels has a limited capacity when compared with a LPNN. A single kernel is about a neural network with only one unit across all hidden layers. The model with multiple kernels has multiple hidden units, but there is no information exchange between hidden units from different kernels.

Relation with factorization machines. We then discuss the relationship between factorization machines [15] and LPNNs. The conclusion is that the second-order factorization machines are special cases of LPNNs.

Theorem 4.2. *The second-order factorization machines taking the form* $y(\mathbf{x}) = w_0 + \mathbf{w}_1^\top \mathbf{x} + \sum_{i=1}^{d_0} \sum_{j=i+1}^{d_0} \mathbf{v}_i^\top \mathbf{v}_j x_i x_j$ *[15] can be written as a LPNN function.*

Proof. In Appendix D.3 in [3], the function of a degree-2 factorization machine can be computed by

$$FM(\mathbf{x};\mathbf{U},\mathbf{S}) = \frac{1}{2}\left[\mathbf{x}^\top\mathbf{U}\mathbf{S}^\top\mathbf{x} - (\mathbf{x}\odot\mathbf{x})^\top \mathrm{diag}\left(\mathbf{S}\mathbf{U}^\top\right)\right]$$

Then we can set $\mathbf{W}^1$ and $\mathbf{V}^1$ and get $\mathbf{h}^1$ of a LPNN model as follows.

$$\mathbf{W}^1 = \begin{bmatrix}\mathbf{U}\\ \mathbf{I}\end{bmatrix}, \mathbf{V}^1 = \begin{bmatrix}\mathbf{S}\\ \mathbf{I}\end{bmatrix}, \mathbf{h}^1 = \begin{bmatrix}(\mathbf{U}\mathbf{x})\odot(\mathbf{S}\mathbf{x})\\ \mathbf{x}\odot\mathbf{x}\end{bmatrix}$$

Then we add a second layer without activation function: set $\mathbf{W}^2 = [\frac{1}{2}\cdot\mathbf{1}^\top, -\frac{1}{2}\cdot\mathrm{diag}\,(\mathbf{S}\mathbf{U}^\top)]$. $\mathbf{W}^2$ will be a 1-row matrix as the factorization machine has only one output value.

Then we have $\mathbf{W}^2\mathbf{h}^1 = \frac{1}{2}\left[\mathbf{x}^\top\mathbf{U}\mathbf{S}^\top\mathbf{x} - (\mathbf{x}\odot\mathbf{x})^\top\mathrm{diag}\,(\mathbf{S}\mathbf{U}^\top)\right] = FM(\mathbf{x};\mathbf{U},\mathbf{S})$. □

By the definition of factorization machines, it will exclude monomials that contain any variables having exponents more than 1, e.g. x_i^2 or $x_i^2 x_j$. But in LPNN model, those monomials will be included, so it is hard to write high-order factorization machines into the LPNN form. However, it is possible to construct a LPNN to match all coefficients of monomials exist in the factorization machines. In this sense, factorization machines have less model capacity than LPNNs.

Relation with tensor-train models. We find that the coefficients of a LPNN has a tensor-train decomposition [14], which means LPNN is a special case of the tensor-train model [5].

Theorem 4.3. *The coefficients of LPNN's network function has a tensor-train decomposition.*

Proof. To simplify the formula, we write $\mathbf{W}^1_{i_1}\mathbf{x} = \sum_{i_0=1}^{d_0}\mathbf{W}^1_{i_1,i_0}\mathbf{W}^0_{i_0}\mathbf{x}_{i_0}$ with $\mathbf{W}^0$ being an identity matrix. Then (11) can be written as

$$\mathbf{h}^L_{i_L} = \sum_{i_{L-1}=1}^{d_{L-1}}\cdots\sum_{i_1=1}^{d_1}\sum_{i_0=1}^{d_0}\left[\left(\mathbf{W}^0_{i_0}\mathbf{x}\right)\prod_{\ell=1}^{L}\mathbf{W}^\ell_{i_\ell,i_{\ell-1}}\mathbf{V}^\ell_{i_\ell}\mathbf{x}\right].$$

Then we can construct a sequence of tensor train cores $\mathbf{G}^0, \mathbf{G}^1, \ldots, \mathbf{G}^\ell, \ldots, \mathbf{G}^L$: $\mathbf{G}^\ell(i_\ell, n_\ell, i_{\ell-1}) = \mathbf{W}^\ell_{i_\ell, i_\ell - 1}\mathbf{V}^\ell_{i_\ell, n_\ell}$ is a three-way tensor for $\ell = 1, \ldots, L-1$. $\mathbf{G}^L$ is a three-way tensor with size 1 in the first dimension, and $\mathbf{G}^L(1, n_L, i_{\ell-1}) = \mathbf{W}^L_{i_L, i_{L-1}}\mathbf{V}_{i_L, n_L}$. $\mathbf{G}^0$ is a three-way tensor with size 1 in the last dimension, and $\mathbf{G}^0(:, :, 1) = \mathbf{I}$. Then $\mathbf{h}^L_{i_L} = \prod_{\ell=0}^{L}(\mathbf{G}^\ell \times_2 \mathbf{x})$. Here $\times_2$ is the 2-mode product of a three-way tensor and a vector; the product represents matrix multiplications. By Lemma 1 in [5], the coefficients of $\mathbf{h}^L_{i_L}$ is the tensor-train decomposition expressed by $\mathbf{G}^0, \ldots, \mathbf{G}^L$. □

In this decomposition, each tensor $\mathbf{G}^\ell$ with $\ell \geq 1$ is constructed from two matrices, which means the tensor-train decomposition has a higher expressiveness than LPNN. Or we can say LPNN is special case of the tensor-train model.

4.2 Special properties

In this section we will discuss two interesting properties of LPNN, showing the advantages of polynomial functions.

Multilinear in parameters The network function $\text{LPNN}(\mathbf{x}; \theta)$ is multilinear in model parameters $\theta = \{\mathbf{W}^1, \ldots, \mathbf{W}^L, \mathbf{V}^1, \ldots, \mathbf{V}^L\}$. The network function (11) shows that all parameters in LPNNs are with an exponent of 1. Because the network function is a polynomial, it is linear in one parameter if we hold all other parameters fixed. This means the function is linear in each variable separately, so it is multilinear. In fact, it is not only multilinear in separate single parameters, but also in separate weight matrix(a $\mathbf{W}^\ell$ or a $\mathbf{V}^\ell$). This property means that the change of a single weight matrix linearly change the network output, which is not held by a normal feedforward network with activation functions like ReLU, sigmoid or quadratic.

The multilinear is a good property in optimization. If we use a convex loss function $\text{loss}(\mathbf{h}^L; y)$ to train a LPNN model, the loss is multiconvex in θ. This property means that the change of a single $\mathbf{W}^\ell$ linearly change the network output. And as a polynomial function, LPNN has a gradient with

simple form. Therefore, in the training process, the gradient of the loss function with respect to the weights θ will have a clear form as the result of back-propagation.

The network function along a gradient . The network function along one direction is a univariate polynomial function, and we can write its canonical form.

In this part we would like to consider the model function when $\mathbf{x}$ is restricted to a line $\mathbf{x} = t\mathbf{g} + \mathbf{x}_0$, where t is a scalar variable, $\mathbf{x}_0 \in \mathbb{R}^{d_0}$ is a point, and $\mathbf{g} \in \mathbb{R}^{d_0}$ is a vector indicating a direction of changing $\mathbf{x}$. Then the model function $\mathrm{LPNN}(\mathbf{x}; \theta)$ is a function of t.

To show the canonical form of this function, we need to find the type of this function. As $\mathbf{x}$ is a linear function of t and $\mathrm{LPNN}(\mathbf{x}; \theta)$ is a polynomial function of $\mathbf{x}$. Then we know the model function will be a univariate polynomial function of t.

To compute the coefficients of this polynomial, we use a recursive method. We define $\alpha(\mathbf{h}^\ell)$ as an operation to extract polynomial coefficients. For example, let $\alpha(ax^2 + bx + c) = [a, b, c]$. Because there are d_ℓ entries in $\mathbf{h}^\ell$ and each entries is a $(\ell + 1)$-order polynomial, which means it has $(\ell + 2)$ polynomial coefficients. Then we have $\alpha(\mathbf{h}^\ell)$ is a matrix with d_ℓ rows, and each row is a vector with $(\ell + 2)$ entries, $\alpha(\mathbf{h}^\ell) \in \mathbb{R}^{d_0 \times (\ell+2)}$. Note that $\alpha(\cdot)$ is a linear operation.

For the base cases, We have $\alpha(\mathbf{h}^0) = \alpha(\mathbf{x}) = [\mathbf{g}, \mathbf{x}_0]$. Then we can calculate $\alpha(\mathbf{h}^\ell)$ recursively. Substitute the expression into (6):

$$
\begin{aligned}
\mathbf{h}^\ell &= \mathbf{W}^\ell \mathbf{h}^{\ell-1} \odot (\mathbf{V}^\ell \mathbf{x}) \\
&= \mathbf{W}^\ell \mathbf{h}^{\ell-1} \odot [\mathbf{V}^\ell (t\mathbf{g} + \mathbf{x}_0)] \\
&= \mathbf{W}^\ell \mathbf{h}^{\ell-1} \odot (\mathbf{V}^\ell \mathbf{g}) t + \mathbf{W}^\ell \mathbf{h}^{\ell-1} \odot (\mathbf{V}^\ell \mathbf{x}_0) \\
&= \mathrm{diag}(\mathbf{V}^\ell \mathbf{g}) \mathbf{W}^\ell \mathbf{h}^{\ell-1} t + \mathrm{diag}(\mathbf{V}^\ell \mathbf{x}_0) \mathbf{W}^\ell \mathbf{h}^{\ell-1}.
\end{aligned}
$$

$\mathrm{diag}(\cdot)$ is a operator to construct a diagonal matrix from a vector.

We have two rules of $\alpha(\cdot)$ operator. If $\mathbf{f}$ is a vector-valued polynomial function of t, and $\mathbf{W}$ is a constant matrix with proper sizes, then we have

$$\alpha(\mathbf{W}\mathbf{f}) = \mathbf{W}\alpha(\mathbf{f}), \qquad \alpha(\mathbf{f}t) = [\alpha(\mathbf{f}), \vec{\mathbf{0}}]$$

Here $\vec{\mathbf{0}}$ is zero vector with the same length as $\mathbf{f}$. The first rule is from the linearity of $\alpha(\cdot)$, and the second rule is can be derived by consider coefficients of each term in $\mathbf{f}t$ and $\mathbf{f}$.

By applying these two rules, we have the recursive formula for computing $\alpha(\mathbf{h}^\ell)$:

$$\alpha(\mathbf{h}^\ell) = \alpha(\mathrm{diag}(\mathbf{V}^\ell\mathbf{g})\mathbf{W}^\ell\mathbf{h}^{\ell-1}t + \mathrm{diag}(\mathbf{V}^\ell\mathbf{x}_0)\mathbf{W}^\ell\mathbf{h}^{\ell-1}) \tag{14}$$

$$= \alpha(\mathrm{diag}(\mathbf{V}^\ell\mathbf{g})\mathbf{W}^\ell\mathbf{h}^{\ell-1}t) + \alpha(\mathrm{diag}(\mathbf{V}^\ell\mathbf{x}_0)\mathbf{W}^\ell\mathbf{h}^{\ell-1}) \tag{15}$$

$$= [\mathrm{diag}(\mathbf{V}^\ell\mathbf{g})\mathbf{W}^\ell\alpha(\mathbf{h}^{\ell-1}), \mathbf{0}] + [\mathbf{0}, \mathrm{diag}(\mathbf{V}^\ell\mathbf{g})\mathbf{W}^\ell\alpha(\mathbf{h}^{\ell-1})]. \tag{16}$$

Then we have the all coefficients in the polynomial function we need. This property is useful when we need to apply perturbations to a input $\mathbf{x}_0$ and compute the effect. For example, in adversarial learning, the generation of adversarial samples often relies on the optimization of a perturbation of an instance $\mathbf{x}_0$. If we know a perturbation direction, then finding the optimal perturbation along the direction is equivalent to finding the minimum of a univariate polynomial function, which can be solved efficiently.

4.3 Training with batch normalization and dropout

As a layer network, a LPNN can be trained with multiple standard techniques including stochastic optimization, batch normalization (BN), and dropout, which have been proved to be effective in practice. Because the product activation function is the only difference between a normal neural network and LPNN, we can apply batch normalization and dropout to a LPNN without any

modification. Here we put the BN layer after the activation.

Let's omit layer indices for notational simplicity and consider one hidden layer in a LPNN model. Let $\mathbf{h}_k$ be the hidden layer of an instance k in a batch, then the hidden layer with batch normalization $\bar{\mathbf{h}}_k$ is computed by

$$\bar{\mathbf{h}}_k = \gamma(\mathbf{h}_k - \boldsymbol{\mu})/(\boldsymbol{\sigma} + \epsilon) + \beta. \tag{17}$$

Here the division / is an element-wise division, ϵ is a small positive number, and (γ, β) are learnable parameters in batch normalization. $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ is the mean vector and variance vector of the training batch. Note that these parameters will not be changed once the training is over, so $\bar{\mathbf{h}}_k$ is a linear function of $\mathbf{h}_k$. As result, a trained LPNN model with batch normalization is still a polynomial function.

Dropout can also be applied to LPNN directly. In the training phase, using dropout is equivalent to removing some entries in summations of (11) and rescaling the summation. In the testing phase, dropout have no effect, and the trained model is just as the definition above. In both situation, the model is polynomial.

4.4 Moments of network outputs in Bayesian learning

Bayesian neural networks(BNNs) combine neural network with Bayesian learning, and provides a way to model uncertainty in a neural network. But it is not trivial to compute the distribution of network outputs as the network parameters are from a distribution in BNNs. In a LPNN model, it will be more convenient to compute the distribution as LPNN is a multilinear in its parameters.

In Bayesian learning, we mainly consider the distribution $p(y|\mathbf{x})$:

$$\begin{aligned} p(y|\mathbf{x}) &= \int_{\theta} p\left(y \mid \mathbf{h}_L = \mathrm{LPNN}(\mathbf{x};\theta)\right) p(\theta)\, \mathrm{d}\theta \\ &= \int_{\mathbf{h}^L} p\left(y|\mathbf{h}_L\right) p(\mathbf{h}^L)\, \mathrm{d}\mathbf{h}^L \end{aligned}$$

$p(\theta)$ is the distribution of parameters, which is either a prior or a distribution inferred from the data. $p(\mathbf{h}_L)$ is necessary to compute the integral. For now, we will use a Gaussian distribution to approximate $p(\mathbf{h}_L)$, which can be decided by the moments of $\mathbf{h}_L$.

In this part, assume the prior distribution of $p(\theta)$ has all matrices independent. This assumption is reasonable because the independent Gaussian distribution is one of the most common choice in Bayesian neural networks[4].

Note that $\mathrm{LPNN}(\mathbf{x};\theta)$ is multilinear in θ. For the first moment of $\mathbf{h}_L$, we have

$$\boldsymbol{\mu}^L = \mathbb{E}_\theta\left[\mathrm{LPNN}(\mathbf{x};\theta)\right] = \mathrm{LPNN}\left(\mathbf{x}; \mathbb{E}_\theta\left[\theta\right]\right). \tag{18}$$

About the second moment, we devenlop a recursive computation method. Let the second-order moments of ℓ-th layer $\mathbf{h}^\ell$ is $\boldsymbol{\Sigma}^\ell = \mathbb{E}_\theta\left[\mathbf{h}^\ell(\mathbf{h}^\ell)^\top\right]$ for $\ell = 1, \ldots, L$. And let the base case $\boldsymbol{\Sigma}^0 = \mathbf{x}\mathbf{x}^\top$. Then the recursive computation is

$$\Sigma_{ij}^\ell = \mathbb{E}_\theta\left[h_i^\ell h_j^\ell\right] = \left(\mathbf{x}^\top \mathbb{E}_\theta\left[(\mathbf{V}_i^\ell)^\top \mathbf{V}_j^\ell\right]\mathbf{x}\right) \cdot \mathrm{tr}\left(\mathbb{E}_\theta\left[(\mathbf{W}_i^\ell)^\top \mathbf{W}_j^\ell\right]\boldsymbol{\Sigma}^{\ell-1}\right) \tag{19}$$

$\mathrm{tr}\left(()\cdot\right)$ means the trace of a matrix.

Now we have both mean and variance of $\mathbf{h}^\ell$, which means we can approximate the distribution of $\mathbf{h}^L$ by a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}^L, \boldsymbol{\Sigma}^L - \boldsymbol{\mu}^L(\boldsymbol{\mu}^L)^\top)$.

Table 1: Approximate the product activation with a one-layer feedforward neural network .

# hidden	1	2	3	4
product	1.13 ± .00	0.80 ± .04	0.33 ±.10	0.03 ±.01
ReLU	0.12 ± .00	0.09 ± .00	0.05 ±.00	0.04 ±.01

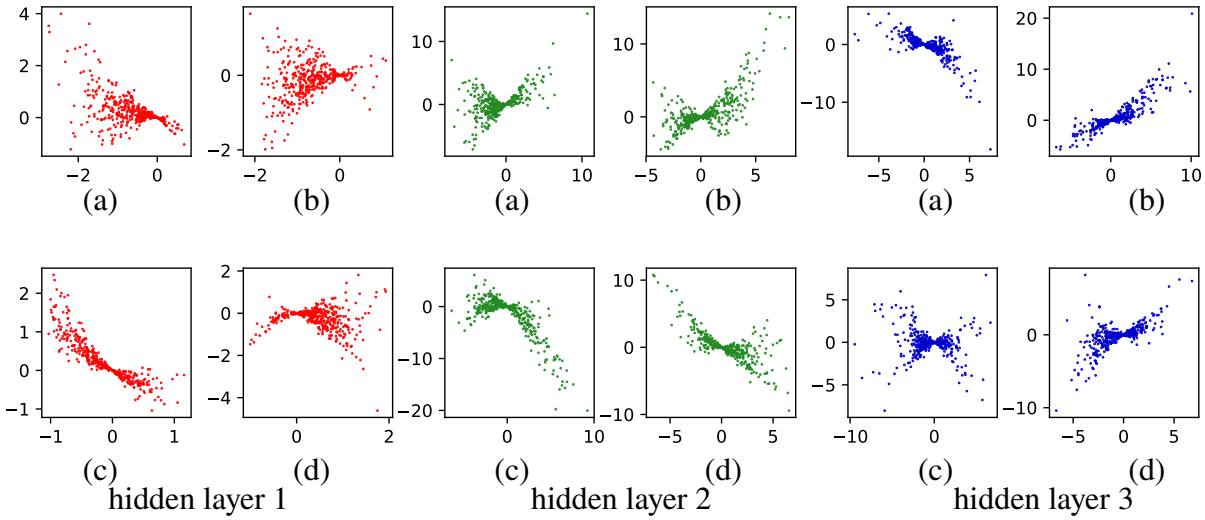


Figure 2: Product activations of LPNN on the mnist dataset. The model has three hidden layers. From each layer, activations of four hidden units are plot here in the same color.

The distribution of network predictions $p(y|\mathbf{x})$ is computable when we have an approximation of $p(\mathbf{h}^L|\mathbf{x})$. In a regression problem, if we assume a Gaussian distribution for $p(y|\mathbf{h}^L)$, the marginal distribution $p(y|\mathbf{x})$ has a closed-form approximation. In a binary classification problem where $y = 1[\mathbf{h}^L > 0]$, then approximation of $p(y = 1|\mathbf{x})$ is $\Phi(\mathbf{h}^L/\sqrt{var(\mathbf{h}^L)})$, with $\Phi(\cdot)$ being the probability mass function(pmf) of the standard Gaussian.

5 Experiment

5.1 The product activation

In this section we would like to discuss the uniqueness of the product activation. Comparing with the normal activation functions, the product activation is a function of not only the layer output$\mathbf{h}$, but

also raw input $\mathbf{x}$. As result, the product activation will have a different function space and cannot be replaced easily. To show this, we perform a experiment to let a normal feedforward neural network to fit a product activation function.

For the setting of the experiment, we randomly generate two features x_1 and x_2 from a uniform distribution $U(-1, 1)$ as input. And we set a product activation function: $y_p = 4x_1x_2$ as the fit target. Now we have the both input and output data for the feedforward neural network to fit.

In this experiment, we use a single hidden feedforward neural network with tanh activation: $y_f = \mathbf{w}^\top \tanh(\mathbf{W}[x_1, x_2]^\top + \mathbf{b}_1) + b_2$. Let l_1 is the number of hidden units, we have $\mathbf{W} \in \mathbb{R}^{l_1 \times 2}$ and $\mathbf{w}^\top \in \mathbb{R}^{1 \times l_1}$. We repeat the experiment with different numbers of hidden units to measure the difficulty in the function fitting. Then we train this neural network with the generated data for 200 epoches. The loss function and evaluate metric is the root-mean-square deviation(RMSE). We have 4 different numbers of hidden units: 1,2,3,4. For each number, we repeat the each experiment for 10 times and take the average and variance of 10 RMSE values.

We also perform the same experiment on another target activation function ReLU as a reference. The ReLU activation is incorporated into a function $y_r = \text{relu}(0.5x_1 + 1.5x_2)/C$, with C normalizing y_r to have an average absolute value of 1, which is same with the average absolute value of product activation function output.

The results of the experiment are tabulated in the Table 1. The first row is the result of fitting product activation function and the second row is about fitting ReLU activation. The result suggests that the feedforward has a relatively large error and can't fit the product activation function very well, especially when the number of hidden units is low. Then the result of fitting ReLU function is listed in the second row, showing that the feedforward network has much smaller error when fitting the ReLU function. So We conclude that the product activation function cannot be replaced easily by other activations like tanh.

We then examine the product activation function $\mathbf{h} = \sigma_p(\mathbf{u}; \mathbf{V}, \mathbf{x})$ in a trained model. We set up a

Table 2: RMSE of different models on regression tasks

methods	wine red	power plant	kin8nm	boston housing	concrete
FF-RELU	0.60 ± 0.04	4.02 ±0.18	0.100 ± 0.002	2.82 ± 0.76	5.10 ±0.49
FM	**0.73 ± 0.09**	4.43 ± 0.15	0.155 ± 0.004	4.80 ± 1.14	8.52 ±0.59
PK	4.39 ± 5.50	**4.14 ± 0.14**	**0.100 ± 0.005**	41.9 ± 77.2	7.95 ±2.42
FF-QUAD	5.49 ± 16.5	5.83 ± 1.39	**0.102 ± 0.007**	**4.59 ± 2.74**	5.58 ±0.48
LPNN	**0.82± 0.18**	**4.24 ± 0.18**	**0.099 ± 0.006**	**4.05 ± 2.13**	**5.20 ±0.62**

Table 3: Error rates of different models on classification tasks

methods	mnist	fashion-mnist	skin	sensIT	letter	covtype-b	covtype
FF-RELU	0.0185	0.108	0.0313	0.176	0.096	0.113	0.146
FM	0.0573	0.167	0.0439	0.260	0.546	0.208	0.575
PK	0.0506	0.168	0.0039	0.225	0.248	0.191	0.494
FF-QUAD	0.0503	0.127	**0.0018**	0.199	0.104	**0.097**	**0.103**
LPNN	**0.0171**	**0.117**	**0.0017**	**0.175**	**0.0729**	0.117	0.140

LPNN model then train it on the MNIST dataset.

MNIST is a dataset of handwritten digits grayscale images with (28×28) pixels. The input feature of each image is a (28×28) matrix contains grayscale value. As LPNN models only accept vectors as inputs, we flatten instances in the MNIST to feature vectors with a length of 784. The number of epochs to train is 20 and the validation accuracy is 0.984.

Then we visualize the product activation by plotting the input $\mathbf{u}$ and the corresponding output $\mathbf{h}$ in Figure 2. Each of the subplot shows the behavior of product activation on a single hidden unit in the model. For each hidden unit, we randomly select 400 instance and plot the (h_i, u_i) pair.

In results, we find that the product activation is not a function of $\mathbf{h}$ as the output value also depends on the input feature $\mathbf{x}$, which means the same input h_i may have different responses. It is not a linear function and the behavior is versatile. But we can see some pattern of quadratic functions in the scatter plot.

5.2 Regression and Classification

In this section, we evaluate the performance of LPNN models on several regression and classification tasks. Our model is compared against the feedforwrad network and three other polynomial learning models. All models are summarized below. For each model, we tune the hyperparameters and choose the optimal setting to make sure the comparison is fair.

Feedforward network (FF-RELU): a classic feedforward network made of fully connected layers, we choose ReLU function as the activations and apply l-2 norm regulraization to the model. The regularization weight is chosen from {1e-6, 1e-5, 1e-4, 5e-4}. When dropout is applied, the dropout rate is chosen from {0, 0.05, 0.1, 0.2, 0.4}.

Polynomial Neural Network (FF-QUAD): a model is the same as the FF-RELU except its activation functions are the quadratic function. It has the same hyperparameters as FF-RELU, and it is trained in the same way.

Factorization Machine (FM): the factorization machine model. We use the implementation from the polylearn library in `sklearn` package [13]. The hyperparameters of FM include the order, the number of factors, and the weight of regularization. The order of FM in this implementation can be 2 or 3. The number of factors is chosen from from {2, 4, 8, 16}, and the regularization weight is chose from the same range as FF-RELU. In the practice, as this implementation of FF-RELU is a binary classifier, it cannot solve the multiclass classification tasks. So we use one-vs-the-rest strategy to deal with this problem.

Polynomial Kernel (PK): the polynomial kernels model. We can specify the order of the underlying polynomial function of PK. The hyperparameters of PK are the same as FM. The implementation is also from the `sklearn` package.

LPNN: our LPNN model, which is the same as the FF-RELU except its activations are product activations. Its hyperparameters are the same as FF-RELU, and it is trained in the same way as

FF-RELU.

We test these models on five regression datasets (wine-quality, power-plant, kin8nm, boston-housing, and concrete-strength) and six classification datasets (mnist, fashion-mnist, skin, sensIT, letter, covtype-b, and covtype). The mnist and fasion-mnist datasets come with the Keras package, the skin, sensIT, and covtype-b datasets are from the libSVM website, and all other datasets are from the UCI repository.

For the regression task, we use that same data split method and experiment process introduced by Yarin Gal [7]. We randomly split each dataset into training set and test set. On each split, we tune the hyperparameters through a five-fold cross validation on the training set to find the optimal setting for each model. Then we train the model with optimal hyperparameters on the whole training set and test it on the test set. The final performance is averaged on 20 different random train-test splits of the data.

For FF-RELU, FF-QUAD, and LPNN, we set three hidden layers and 50 hidden units in each hidden layer. We apply both dropout and batch normalization to all the three models. The LPNN model with three hidden layers has a polynomial order of 4, so we set the same order for the PK model. We set the order of FM to be 3 as it cannot have a higher order in the implementation. For each model, we tune all hyperparameters described in the aforementioned range.

Table 2 tabulates the experiment result. It shows the average RMSE over 20 splits and its standard deviation. And we compare LPNN against the competing polynomial models with paired t-tests, and bolded one or more the best performance of the polynomial model. We also list the performance of FF-RELU as reference. In general, LPNN performs better than other polynomial models. FF-QUAD is not stable when its polynomial order is high, as the bad performance on the wine-quality dataset shows. The performance of LPNN is slightly worse than the performance of FF-RELU.

We then test these models on seven classification tasks. For each dataset, we randomly select 30% of the data as the test set, except for mnist and fasion-mnist datasets, whose test sets are provided.

To do the model selection for both architecture and hyperparameters, we use 20% of the training set as the validation set. For the model architecture, the number of hidden layers is chosen from $\{1, 2, 4\}$. Then we apply a reversed pyramid network structure which the number of hidden units in each layer shrink as the network becomes deeper. The number of hidden units at ℓ-th layer is computed by $\alpha^{\ell}(d_{out} - d_{in}) + d_{out}$, where d_{in} is the number of input feature, d_{out} is the number of output unit, and the alpha is the shrinking factor. We select α from $\{0.3, 0.5, 0.7, 0.8\}$. We also select the order for PK from $\{2, 3, 5\}$ to match the order of LPNN. All other hyperparameters of a model are also selected together with architectures in the same range listed in the previous section.

The error rates of different models are reported in Table 3. The performance of the best polynomial model is bolded. We also compare LPNN against FF-RELU: if LPNN is significantly better than FF-RELU, we underline the LPNN's performance. The performance of LPNN is better other polynomial models in general. And LPNN is comparable to FF-RELU on classification tasks.

We notice that the LPNN model has a better performance on classification tasks. We speculate the reason is the restricted function form of the LPNN. As a polynomial model, an LPNN can only have a model function as a polynomial, which limits its ability to fit the exact value in regression tasks, especially when the target real-world function is not a polynomial. In classification tasks, the model only needs to decide discrete labels not the exact value. LPNN may be not flexible enough for fitting continuous values from arbitrary function compared to feedforward networks.

But still, the LPNN narrows the performance gap between polynomial models and the well-studied feedforward neural networks, making the polynomial more practical in typical learning tasks.

5.3 The effect of batch normalization and dropout

In this subsection, we investigate the effects of batch normalization and dropout on LPNN. It is important to know if those effective methods can be applied to LPNN models. We test the LPNN

Table 4: Effect of batch normalization and dropout on the concrete-strength dataset

L	1	2	3	5	10
neither	**7.76 ± 0.53**	**6.30 ± 0.96**	6.89 ± 2.06	7.86 ± 3.10	7.49 ± 2.77
only dropout	**7.78 ± 0.54**	**5.98 ± 0.57**	**5.12 ± 0.58**	**4.92 ± 0.78**	**4.71 ± 0.90**
only BN	**7.82 ± 0.55**	**6.26 ± 0.87**	5.82 ± 1.13	**5.46 ± 1.83**	**4.97 ± 0.97**
BN and dropout	**7.77 ± 0.53**	**6.05 ± 0.50**	**5.20 ± 0.62**	**4.72 ± 0.66**	**4.58 ± 0.74**

Table 5: Effect of dropout and BN on the mnist dataset

L	1	2	3	5	10
neither	**0.0171**	**0.0192**	**0.0207**	0.8947	**0.0241**
only dropout	0.0208	**0.0188**	**0.0187**	0.7657	0.0298
only BN	0.0242	**0.0202**	0.0229	0.0271	**0.0207**
BN and dropout	**0.0191**	**0.0191**	**0.0170**	**0.0207**	**0.0230**

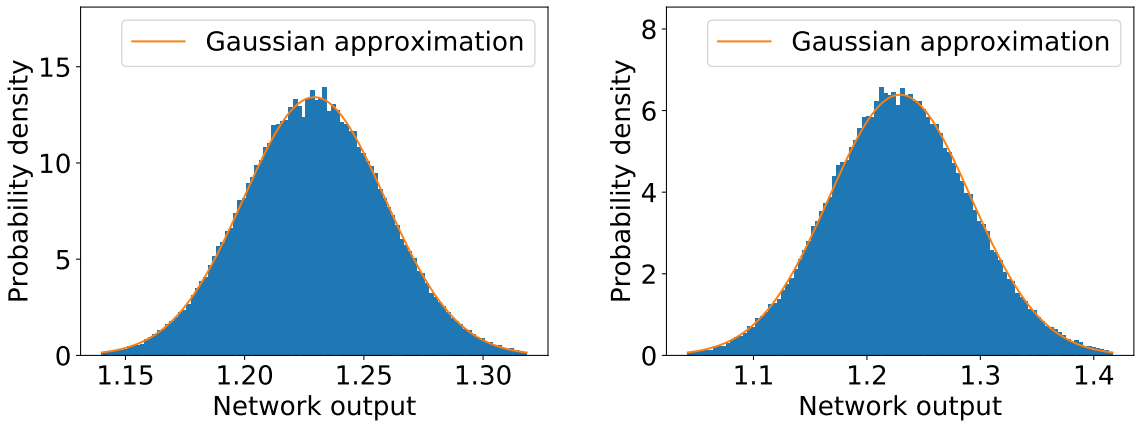
models with different depth: $L \in \{1, 2, 3, 5, 10\}$. For each depth, we try four combinations: using neither batch normalization nor dropout, using dropout only, using batch normalization only, using both methods. We still select other hyperparameters through model selection. We run the experiment on a regression task (concrete-strength) and a classification task (mnist).

The results are shown in Table 4 and 5. For each depth L, the four combinations are compared, and the best performance(s) across four combinations are bolded. From this result, we see that both batch normalization and dropout are needed to train a good LPNN model when the model is deep. On the mnist dataset, the LPNN without batch normalization has very bad performance when $L = 5$. Its performance drops sharply after a few epochs. This observation indicates that the LPNN without batch normalization is very unstable due to some bad optimization directions.

5.4 Network outputs with stochastic network weights

In this section, we show the applicaiton of LPNN model in Bayesian learning. In the previous section we discuss the method to approximate the distribution of a Bayesian LPNN output as the network parameters is from a Gaussian prior. In this experiment we check the network output $\mathbf{h}^L$

Figure 3: The distribution of the network outputs.



of a L-layer LPNN model when the network parameters θ are from a Gaussian prior . We use a LPNN with $L = 5$ layers. The mean of the prior is given by network weights of a trained LPNN for a regression task, which means the network output $\mathbf{h}^L$ has only one entry. The variance of the prior is set to $\sigma^2\mathbf{I}$ and we show the result of different σ. Then we sample 10,000 samples from the Gaussian prior as network weights to compute samples of $\mathbf{h}^L$. Then we can plot its samples into a histogram. At the same time, following the method in preivous section, we compute the first and second moments of the distribution of $\mathbf{h}^L$ and get a Gaussian approximation. Figure 3 shows the histograms and the corresponding approximate Gaussian distributions for $\sigma^2 = 0.05$ (left) and $\sigma^2 = 0.1$ (right). From this result, we see that the Gaussian approximation is very accurate.

6 Conclusion

In this paper, In this paper, we have proposed a new activation function: product activation function. This activation function multiplies a linear transformation of the input to the hidden layer to achieve a non-linear activation function. It can be used as a normal activation functions in a neural network. The product activation always increases the polynomial order by 1. By using product activation in a feedforward neural network, we obtain the Ladder Polynomial Neural Network (LPNN). LPNN is a new type of polynomial neural networks that can have an arbitrary polynomial order.

LPNN has a polynomial model function and benefits from multiple good properties of polynomial functions. We discuss and prove some of those properties. Then we perform multiple experiments to verify those advantage in practice. We also test our model in both classification and regression tasks on various datasets. The results show that LPNN outperforms other polynomial models in those datasets and is approaching the normal feedforward neural networks.

At the same time, LPNN can also be trained with modern training techniques like dropout and batch normalization, as the experiment shows. The LPNN has the potential to provide new insights to the theoretical study of polynomial learning models and. It is also a valuable learning method in practice.

References

[1] Mathieu Blondel et al. "Higher-order factorization machines". In: *Advances in Neural Information Processing Systems*. 2016, pp. 3351–3359.

[2] Mathieu Blondel et al. "Multi-output polynomial networks and factorization machines". In: *Advances in Neural Information Processing Systems*. 2017, pp. 3349–3359.

[3] Mathieu Blondel et al. "Polynomial Networks and Factorization Machines: New Insights and Efficient Training Algorithms". In: *International Conference on Machine Learning*. 2016, pp. 850–858.

[4] Charles Blundell et al. "Weight uncertainty in neural networks". In: *arXiv preprint arXiv:1505.05424* (2015).

[5] Z. Chen et al. "Parallelized Tensor Train Learning of Polynomial Classifiers". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.10 (Oct. 2018), pp. 4621–4632.

[6] Simon S Du and Jason D Lee. “On the Power of Over-parametrization in Neural Networks with Quadratic Activation”. In: *International Conference on Machine Learning*. 2018, pp. 1328–1337.

[7] Yarin Gal. `https://github.com/yaringal/DropoutUncertaintyExps`. Accessed in 2019.

[8] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015).

[9] Joe Kileel, Matthew Trager, and Joan Bruna. “On the Expressive Power of Deep Polynomial Neural Networks”. In: *arXiv preprint arXiv:1905.12207* (2019).

[10] Ming Lin et al. “The second order linear model”. In: *arXiv preprint arXiv:1703.00598* (2017).

[11] Zachary C Lipton. “The Mythos of Model Interpretability: In machine learning, the concept of interpretability is both important and slippery.” In: *Queue* 16.3 (2018), pp. 31–57.

[12] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. “On the computational efficiency of training neural networks”. In: *Advances in neural information processing systems*. 2014, pp. 855–863.

[13] Vlad Niculae. `https://github.com/scikit-learn-contrib/polylearn`. Accessed in 2019.

[14] I. V. Oseledets. “Tensor-Train Decomposition”. In: *SIAM Journal on Scientific Computing* 33.5 (2011), pp. 2295–2317.

[15] Steffen Rendle. “Factorization machines”. In: *2010 IEEE International Conference on Data Mining*. IEEE. 2010, pp. 995–1000.

[16] Mohammadreza Soltani and Chinmay Hegde. “Fast and Provable Algorithms for Learning Two-Layer Polynomial Neural Networks”. In: *IEEE Transactions on Signal Processing* 67.13 (2019), pp. 3361–3371.

[17] Mohammadreza Soltani and Chinmay Hegde. “Towards provable learning of polynomial neural networks using low-rank matrix estimation”. In: *International Conference on Artificial Intelligence and Statistics*. 2018, pp. 1417–1426.

[18] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[19] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. 2017, pp. 5998–6008.