This a preprint that was to Pattern Recognition Letters and is NOT peer-reviewed. However, they didn’t publish it (but saw no problems in its scientific merits) but recommended submitting it to a journal focused on neural networks instead.

Pattern Recognition Letters
journal homepage: www.elsevier.com

Extending Neural Networks Using Polynomial Arithmetic

Tomas Ukkonen

ABSTRACT

The multilayer neural network architecture has seen relatively little number-theoretic research. However, improving fundamental computing capacity by extending to the more extensive set of possible numbers and functions could improve machine learning results. Networks often use real and sometimes complex numbers, and extending numbers to contain a more rich structure may improve the abstract processing of real-world data. A polynomial ring can generalize multiplication to convolution while the neural network is as close as possible to the standard one. Experiments using synthetic and cryptographic problems compare performance between real-valued standard and polynomial ring implementations. However, results show slightly larger errors when using polynomial ring neural networks.

1. Introduction

Number theoretic research in neural networks research has not been very common although more complex computations and functions could improve optimization results. This happens without the need to do significant modifications to the standard architectures (fully connected, recurrent) or core algorithms (backpropagation, gradient descent, activation functions) (Haykin, 1999). In research, neural networks have been typically only extended to process complex numbers or sometimes quaternions (Nitta, 1993; Tachibana & Otsuka, 2018; Hirose, 2003; Gaudet & Maida, 2018).

2. Theoretical background

This paper's motivation to extend real numbers to a polynomial ring is an intuition that real numbers can be extended to contain dimensionality information when using, for example, fractals (Falconer, 2003). A quantity of dimension d can be the multiplication of a real-valued scalar a and a volume of d-dimensional hypercube r^d, where r is the arbitrary (unit) length of one side of the hypercube. One can then make a somewhat simple definition of "multidimensional" numbers s.

$$s = a_0 * r^0 + a_1 * r^1 + a_2 * r^2 ... \quad (1)$$

Compared to the magnitude of real numbers, r's length is infinite and outside of the set $\mathbb{R}$, so the components are in the same direction but perpendicular to each other in a vector space sense because there is no scalar $a \in \mathbb{R}$ to scale different components to be the same. It seem to be possible to extend this line of reasoning formally to cases where d is real-valued, meaning fractal dimensions, or abstractly to imaginary and negative dimensions (r^{-1} is infinitesimal dr) and, in addition to multiplication, define a stretching operation c to an object $a_{new} * r^d = a_{old} * (c * r)^d = a_{old} * c^d * r^d$. By incorporating dimensional information into a neural network, it might better abstractly process real-world data.

To have a well-defined closed number system that can compute using computers, dimensions are restricted to be natural numbers $\mathbb{N}$ and a real polynomial ring $R[X]/(X^K - 1)$ (Moore, 2008) is defined using the following definitions (Equation 2).

$$\begin{aligned} s &= \sum_{d=0}^{K-1} a_d * r^d, a_d \in \mathbb{R}, d \in \mathbb{N} \\ s_1 + s_2 &= \sum_{d=0}^{K-1} (a_{d_1} + a_{d_2}) * r^d \\ s_1 * s_2 &= \sum_{d_1=0,d_2=0}^{K-1,K-1} a_{d_1} * a_{d_2} * r^{d_1+d_2 (mod) K} \end{aligned} \quad (2)$$

By choosing modulo operation K to be a prime number, it is easy to see that the polynomial ring becomes a field with a finite K number of components. Multiplication operation now

e-mail: tomas.ukkonen@novelinsight.fi (Tomas Ukkonen)

leads to a circular convolution of the coefficients. The circular convolution and its inverse, the division operation, can be efficiently calculated using discrete Fourier transform (Mitra, 1998). By making dimensions d circular, the comparability of the numbers is lost even when using real coefficients. However, if numbers/the circular convolution is properly zero-padded, it is often possible to calculate the standard convolution. The circular convolution operation processes each component symmetrically, ignoring dimensional information. Asymmetry is later (weakly) imposed by applying stretching operation when initializing the neural network's weights so that weights are approximately zero-padded and in a ReLU activation function which processes the first dimension differently.

3. Neural network architectures using polynomial ring numbers

3.1. Linear model

The simplicity of the linear optimization is that the function $\mathbf{y} = \mathbf{A} * \mathbf{x} + \mathbf{b}$ has only one easily solvable MSE optimum, and it is the global optimum of the problem. In practice, linear functions are often too simple for many practical problems, but if we can do a non-linear number theoretic extension to real numbers fulfilling field axioms, it is possible to solve the global optimum of non-linear problems. Unfortunately, for polynomial field numbers s and real-valued data, the non-real parts of the coefficients are always zero, meaning that it is impossible to extend calculations non-linearly.

In data analytics, however, it is common to discretize real-valued variables using one-hot encoding, which maps real numbers to higher dimensional vectors, after which the global optimum of an approximated problem is solvable. In this paper, other number theoretic possibilities are not studied. A multilayer neural network is used instead, in which non-linearities make it possible to process real-valued data using extended number systems.

3.2. Neural network model

The neural network was kept as close to linear as possible to keep optimization methods like gradient descent functional. The densely connected neural network's layers use a leaky ReLU non-linearity (Hahnloser et al., 2000; Glorot et al., 2011) except at the last layer, which is entirely linear. The ReLU activation function is extended to polynomial fields by applying ReLU non-linearity $f(s)$ based on the value of the first dimension (Equation 3).

$$\begin{aligned} f(s) &= \begin{cases} as, & \text{if } s_0 < 0\,, \\ s, & \text{if } s_0 \geq 0\,. \end{cases} \\ df/ds &= \begin{cases} a, & \text{if } s_0 < 0\,, \\ 1, & \text{if } s_0 \geq 0\,. \end{cases} \end{aligned} \tag{3}$$

Derivates are also well-defined for linear algebra operations when using polynomial field numbers s. This is because derivates are not dependent on what direction zero is approached.

$$df(s)/ds = \lim_{h\to 0}((a*(s+h)+b) - a*s)/h = lim_{h\to 0}a = a \tag{4}$$

With these definitions it is possible to calculate the jacobian matrix of the neural network using backpropagation formulas. To calculate the derivate of mean squared error using polynomial field, MSE is written using an operator $\star$, which multiplies each component of the polynomial field.

$$a \star b = \sum_{d=0}^{K-1} a_d b_d r^d \tag{5}$$

We can then write MSE error as a $\star$ product of $\mathbf{1} = 1 * r^0$, which selects the real part of the error term, and a $\star$ product of error delta terms that squares error values. Equation 6 is for simplicity written using dimensions j but can be generalized to vectors and its derivate calculated from Jacobian matrix.

$$MSE(\mathbf{w}) = \mathbf{1}\star(\frac{1}{2N}\sum_j\sum_{i=0}^{N-1}(f_j(\mathbf{x_i}|\mathbf{w}) - y_{ij})\star(f_j(\mathbf{x_i}|\mathbf{w}) - y_{ij})) \tag{6}$$

Now, because of the close resemblance of our polynomial ring to the Fourier transform, it is possible to transform $\star$ products to convolutions by calculating the Fourier transform $\mathfrak{F}$ of the terms. These convolutions and Fourier transforms can be easily derived because they are not dependent on the neural network's weight parameters $\mathbf{w}$. In gradient equations 7, a $\circ$ operator marks circular convolution operation.

$$\begin{aligned} MSE(\mathbf{w}) &= \mathbf{1} \star MSE'(\mathbf{w}) \\ \frac{dMSE'(\mathbf{w})}{d\mathbf{w}} &= \mathfrak{F}^{-1}(\frac{1}{N}\sum_j\sum_{i=0}^{N-1}\mathfrak{F}(f_j(\mathbf{x_i}|\mathbf{w}) - y_{ij}) \circ \mathfrak{F}(\frac{df_j(\mathbf{x_i}|\mathbf{w})}{d\mathbf{w}})) \\ \frac{dMSE(\mathbf{w})}{d\mathbf{w}} &= \mathfrak{F}^{-1}(\mathfrak{F}(\mathbf{1}) \circ \mathfrak{F}(\frac{dMSE'(\mathbf{w}))}{d\mathbf{w}}) \end{aligned} \tag{7}$$

In experiments, non-real parts of the error vector is always set to be zero. The idea that stretching an object should not change the properties of an object is used to apply stretching operation $c = 0.25$ to initial random weights. This means that larger dimensional weights are initially close to zero, and the neural network processes data in low dimensions. Therefore computations start with zero padding of the convolution operations between numbers but allow the network to use the circular convolution and higher dimensions if it reduces MSE error.

3.3. Gradient descent algorithm

Algorithm 1 describes a modified gradient descent algorithm that trains polynomial ring neural networks. It uses costly adaptive step length and sometimes enters worse solutions. The mechanism allows the algorithm to escape from local minimums into what polynomial ring minimization often gets stuck. The algorithm rarely converges but tries worse solutions looking for a way away from the local minimum.

Algorithm 1 Gradient Descent algorithm for polynomial ring neural network

procedure GRADIENTDESCENT
  $e \leftarrow \infty$
  while $e > error\_limit$ **do**
    $e \leftarrow calculate\_error(w)$
    $g \leftarrow calculate\_gradient(w)$
    $errors \leftarrow \infty$
    $epsilon \leftarrow 0.02$
    $iters \leftarrow 0$
    while $last(errors) > e$ & $iters < 500$ **do**
      $epsilon \leftarrow epsilon/2$
      $w_{next} \leftarrow w + epsilon * g$
      $iters \leftarrow iters + 1$
      $errors \leftarrow calculate\_error(w_{next})$
      $r \leftarrow (random()\%40)$
      if $last(errors) < 2 * average(errors, 20)$ & $r == 0$ **then**
        $errors \leftarrow 0$; goes to worse solution with a 2.5% probability
    $w \leftarrow w_{next}$

4. Experimental results

Experiment 1 uses a synthesized problem and a small number of parameters. The amount of synthetic data is only $N = 1000$, so even our unoptimized algorithm would run experiments quickly. The training did not do early stopping and overfitted to the data. The standard neural network implementation is TensorFlow's with Adam optimizer which is comparated against our gradient descent algorithm. The non-linearities of the first layers of the neural network are ReLU activation functions, and the last layer is linear. The real-valued input variables $\mathbf{x} \sim N(\mathbf{0}, 4^2\mathbf{I})$ are mapped to values $\mathbf{y}$ using the following equations.

$$\begin{aligned} y_1(\mathbf{x}) &= \sin(f * x_1 * x_2 * x_3 * x_4) \\ y_2(\mathbf{x}) &= sign(x_4) * a^{x_1/(|x_3|+1)} \\ y_3(\mathbf{x}) &= sign(\cos(w * x_1)) + sign(x_2) + sign(x_4) \\ y_4(\mathbf{x}) &= x_2/(|x_1|+1) + x_3 * \sqrt{|x_4|} + |x_4 - x_1| \\ f &= 10.0, w = 10.0, a = 1.1 \end{aligned} \tag{8}$$

Results of overfitting optimization are in Table 1. The algorithm computed results five times, and the best value was chosen. The problem was too complicated for a small two-layer $4 - 4 - 4$ neural network, and algorithms couldn't learn the dataset. However, even in this case, standard neural network implementation performed better for normal data than polynomial ring neural networks. This result shows that polynomial ring neural networks still require more work to optimize the non-linearity, the number of dimensions, initialization of weights, and the learning algorithm if they would be usable as general problem solvers in machine learning.

Table 1. Minimum absolute error found with different neural network architectures.

Architecture	Error
4-4-4 (real-valued)	1.0245
4-4-4 (11 dimensional numbers)	1.10242
4-20-20-4 (real-valued)	0.4218
4-20-20-4 (11 dimensional numbers)	0.6310

4.1. Experiment 2

The second experiment attempts to learn the inverse of the SHA-256 cryptographic hash function (Stinson, 2002). The data ($N = 1000$) contains ten first bits of the SHA-256 hash values computed from random ten-letter ASCII messages.

A small neural network was chosen to compare the computing capacity of the different parameterizations. Also in Experiment 2, TensorFlow and the standard ReLU and Adam optimizers were used to optimize a $10 - 20 - 20 - 10$ neural network. These were compared to a linear solution and 11-dimensional numbers $10 - 20 - 20 - 10$ neural network. The minimum mean absolute error found using ten runs is reported in Table 2. The running times of unoptimized code are much longer than the standard TensorFlow implementation, which only takes few minutes to reach its optimum. This is because 11-dimensional numbers use convolution in multiplication, which increses number of multiplications needed and causes, on average, a 100-1000 fold slowdown in multiplication. In theory, Fast Fourier Transform could reduce the number of multiplications from $O(N^2)$ to $O(Nlog(N))$, but this is true only for large values of N.

Table 2. Minimum absolute error found with different architectures.

Architecture	Error
10-20-20-10 (real-valued)	0.226
Linear model (11 dimensional numbers)	0.248
10-20-20-10 (11 dimensional numbers)	0.235

Also here results with the 11-dimensional polynomial ring neural network is slightly worse than with the standard neural network despite the fact that it has more free parameters.

5. Discussion

In this paper, an exciting number theoretic extension to neural networks is described. However, the optimization is plagued by problems such as getting stuck to local optimums, which require using modified gradient descent algorithm. More research is needed to solve these problems and to improve results to be on par or better with the standard neural network.

Although it has not been studied in this paper, the convolutional nature of the number system could make it helpful in the real-world processing of audio and picture signals. Another line of further study would be to try the effect of different nonlinearities to the maximum possible network depth.

6. Funding and further information

This research has not received any funding or grants from any sources. Polynomial ring neural network implementation will be published as open source as part of Dinrhiw C++ machine learning software library (Ukkonen, 2024).

References

Falconer, K. (2003). *Fractal Geometry: Mathematical Foundations and Applications*. (2nd ed.). John Wiley & Sons Ltd.

Gaudet, C. J., & Maida, A. S. (2018). Deep quaternion networks. In *2018 International Joint Conference on Neural Networks*. IEEE. doi:10.1109/IJCNN.2018.8489651.

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks.

Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., & Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, *405*, 947–951.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. (2nd ed.). Prentice-Hall International.

Hirose, A. (2003). *Complex-Valued Neural Networks: Theories and Applications*. World Scientific Publishing Cp. Ptd. Ltd.

Mitra, S. K. (1998). *Digital Signal Processing: A Computer-Based Approach*. McGraw-Hill series in electrical and computer engineering.

Moore, K. B. (2008). *Discrete Mathematics Research Progress*. Nova Science Publishers Inc.

Nitta, T. (1993). A back-propagation algorithm for complex numbered neural networks. In *Proceedings of 1993 International Conference on Neural Networks*. IEEE. doi:10.1109/IJCNN.1993.716968.

Stinson, D. R. (2002). *Cryptography: Theory and Practice*. (2nd ed.). Chapman & Hall/CRC.

Tachibana, K., & Otsuka, K. (2018). Wind prediction performance of complex neural network with relu activation function. In *2018 57th Annual Conference of the Society of Instrument and Control Engineers of Japan*. IEEE. doi:10.23919/SICE.2018.8492660.

Ukkonen, T. (2024). Dinrhiw2: C++ machine learning library. URL: https://github.com/cslr/dinrhiw2.