Self-Adaptive Layer: An Application of Function Approximation Theory to Enhance Convergence Efficiency in Neural Networks

1st Ka-Hou Chan
School of Applied Sciences,
Macao Polytechnic Institute,
Macao, China
chankahou@ipm.edu.mo

2nd Sio-Kei Im
Macao Polytechnic Institute,
Macao, China

3rd Wei Ke
Macao Polytechnic Institute,
Macao, China

***Abstract*—Neural networks provide a general architecture to model complex nonlinear systems, but the source data are often mixed with a lot of noise and interference information. One way to offer a smoother alternative for addressing this issue in training is to increase the neural or layer size. In this paper, a new self-adaptive layer is developed to overcome the problems of neural networks so as to achieve faster convergence and avoid local minimum. We incorporate function approximation theory into the layer element arrangement, so that the training process and the network approximation properties can be investigated via linear algebra, where the precision of adaptation can be controlled by the order of polynomials being used. Experimental results show that our proposed layer leads to significantly faster performance in convergence. As a result, this new layer greatly enhances the training accuracy. Moreover, the design and implementation can be easily deployed in most current systems.**

***Index Terms*—Function Approximation, Orthogonal Polynomial, Self-Adaptive, Neural Network**

I. Introduction

An Artificial Neural Network is an example of information processing model that is inspired by biological nervous systems such as the brain and sense information. The major element of this paradigm is the novel structure of the information processing system. It consists of a large number of highly interconnected processing nodes or neurons for solving specific problems [1]. Making use of neural networks in deep learning can obtain various undiscovered relations between the input and the output of training datasets for dynamic system modeling. In recent years, many researchers have been trying to solve the existing problems of feedforward neural networks. In this type of networks, the information moves in only one direction from the input nodes, forwards through the hidden layers and finally to the output nodes [2]. In this context, many researchers that have been involved in the use of feedforward neural networks for dynamic system control and in the field of system development are making great efforts for the artificial dynamic model of the real process [3].

In order to find the relationship between the input and output nodes, the deep learning process explains that as the black box concept about middle hidden layers. This feature of unknown function is quite similar to function approximation approach [4]. Therefore, some approach of neural networks to be developed is based on orthogonal polynomials that are widely used for unknown function approximation [5]. According to the theory of function approximation, any function $f\left(x\right)$ can be approximated in the form of

$$f\left(x\right)=\sum_{i=0}^{\infty}c_iT_i\left(x\right), \tag{1}$$

where c_i are the coefficients and $T_i\left(x\right)$ are the basis functions. These functions are constructed orthogonally with respect to certain measure. They satisfy the following,

$$\int_a^b T_i\left(x\right)T_j\left(x\right)dx=0 \quad (i\neq j). \tag{2}$$

We can have different series of polynomials by choosing different basis functions. The well-known serieses such as Fourier, Chebyshev and Legendre, *etc.* are all orthogonal polynomials. Neural networks can provide a convenient way of coefficients determination, because they are universal approximators that can learn the data by examples or reinforcement [6], either in the batch or sequential mode. Their expansion forms are independent to one another, if the processing elements of a neural network are composed of the expansion forms of an orthogonal function, the unique set of coefficients of the expansion forms will be the training parameters of the processing elements. It can be easily trained to map nonlinear functions because of their parallel architecture. However, much of the approximation theory does not generalize well to multivariate approximation problems [7]. Because most multivariate approximation problems involve tensor product spaces, they are highly dependent to the basis functions used and require coefficients to increase in geometric progression [8].

A. Related Work

Orthogonality, as the major concept in function approximation, has been attractive in orthogonal architectures. Orthogonal theory accurately preserves the forward and continuous gradient through convergence. It has been found that the

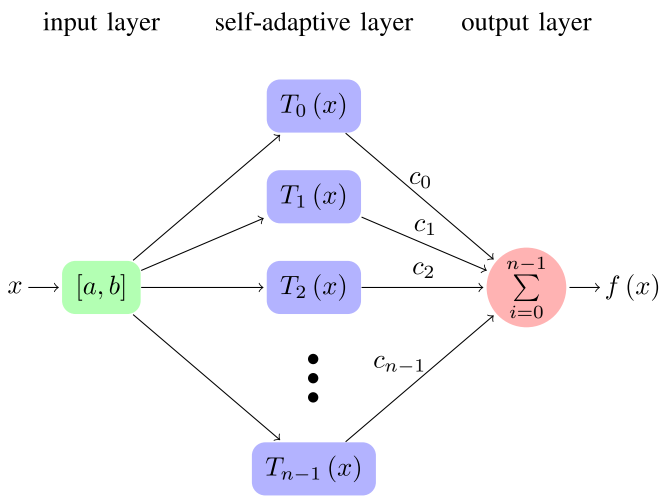

Fig. 1: The single self-adaptive layer which corresponding to (1) and (2).

theory can be used in neural networks to better approximate a wide range of nonlinear results to any desired accuracy [9], [10]. Later, in [5], it introduces the network that attempts to combine the best aspects of both orthogonal function expansion and neural network. The function approximation by using neural networks was presented in [11]. It has been shown to have the function similar to orthogonal polynomial theory. These experimental results showed that orthogonal polynomial functions have less approximation error than traditional methods. A system is developed in [12] for the linear neural networks when the weights are orthogonal initializations, and even independent to the depths, where training time can be shortened and is independent to the sequence size for a fully linear orthogonal neural network. From other entry points, in [13] by using orthogonal random samples, it provides much faster convergence, that improves the convergence properties of the Gaussian kernel approximation over random Fourier features [14]. Further, a nonlinear base function, such as *sine* and *cosine*, as the activation function that is locally orthogonal is proposed in [15], which shows a great potential to have faster convergence [16]. However, most papers consider that nonlinear systems have no constraints and they are just trying some activation functions to approximate those uncertainties for achieving objectives relationship. System performance may be unpredictable by these methods when there are constraints.

Meanwhile, by using adaptive neural network dynamic techniques, several interesting solutions have been presented for nonlinear systems [17], [18]. In [19], the problem of adaptive neural network for a classifier of nonlinear system is dealt with, and in [20], it presents an adaptive network design that is used for the clustering of nonlinear discrete-time systems with web users. Then in [21], an adaptive neural output feedback control design for finite-time nonlinear systems is proposed. However, these designs are used here to refer to a specific neural network, they are challenged to be applied to general problems without expansions or constraints. Furthermore, an efficient layer can achieve a better assembly than the above architectures for different cases. There are various layers that can filter out the noise or useless information form the input samples then leave the main features as output to following processing. In [22], it proposes the convolutional layer that can better solve the problem of image classification, and in [23], it provides the recurrent multi-layer for incorporating temporal dynamics and time series. Their major advantages are used for modularity that is possible to be embedded in all architectures for advanced used. These algorithms have also been implemented into the layer level that can be combined with various layers or activation functions [24], [25]. These are all due to the concept of layers, user can custom the layer size and order inside the neural network architecture [26].

The self-adaptive layer to be presented in this paper is based on the function approximation theory. A complete configuration of this neural network is illustrated in Fig. 1. Each input x will evolve into an output $f(x)$ that provides the enhanced feature for the learning. There are no weights between the input layer and self-adaptive layer because the input will be substituted into these orthogonal polynomials $T_i(x)$. The i-th node of this self-adaptive layer is the i-th polynomial of the series used. We should ensure the condition that the input value must be within the interval $[a,b]$. Because the coefficients c_i of all the terms should be determined in the processing, they will be seen as the training parameters like the weights of a simple linear layer. In other words, the result can keep the advantages of function approximation, such as fast convergence and avoidence of local minimum. By using neural networks, the problem of initial coefficients (weights) does no longer exist. Therefore, the orthogonal neural network is single-layered and linear.

B. Outline

The rest of the paper is organized as follows. Section II discusses the concept of function approximation making use of neural networks. A couple of instances are illustrated to give an analysis of the adaptive of layer use and conditions, including that the effectiveness in adapting can be controlled or overfitting by the size of polynomials used. Next, in Section III, we give the idea of how the new self-adaptive layer can be integrated, followed by the configurations of our experiment and evaluation environment. The results and improvements are illustrated and discussed in detail in this section as well. Finally, Section IV concludes the paper.

II. Function Approximation Theory

For the convolutional and recurrent layers respectively in the Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), the input data passing through these layers have correspondence to the results we are interested. Unfortunately, in general cases, it is hard to discover or develop certain layers that can find the feature information $F(x)$ we want directly from the input data x, together with the mapping $x \to F(x)$. For the models of most dynamic systems, this remains unknown or hard to determine. Therefore, it is necessary to develop a method that can use the input and output data of a system to determine the required layer. To tackle this problem, we apply a single-layer neural network developed according to function approximation theory, and make use of it into our training process. Assume that the unknown function is $F(x)$, the training algorithm uses the feedback of loss to the orthogonal neural network to train these coefficients c_i as follows,

$$F(x) \approx f(x) = \sum_{i=0}^{\infty} c_i T_i(x),$$

where $F(x)$ is the undiscovered algorithm layer whose detail is out of our discussion, $f(x)$ is the approximate function that can be formed by different series of orthogonal polynomial sets, and the coefficients c_i are unique as the training parameters. The major advantage of $f(x)$, in its expansion form, is that the error can be decreased by increasing the degree i of the polynomial $T_i(x)$ used.

A. *Self-Adaptive Layer*

Different from what have been proposed in [5] and [11], we are not going to change the feature size during approximation, and each output feature is just affected by one corresponding input variable and the weights (coefficients) of the orthogonal functions (see Fig. 1). Therefore, the self-adaptive layer can often be achieved along with the input layer directly, without affecting the following architecture. For more general adaptation, we recommend the first kind of Chebyshev series [27] as the basis set of orthogonal polynomials in our implementation, thus polynomial $T_i(x)$ of the i-th degree can be presented as,

$$T_i(x) = \cos(i \arccos(x)), \tag{3}$$

where its interval is $[-1.0, +1.0]$. (1) then becomes

$$f(x) = \sum_{i=0}^{n-1} c_i \cos(i \arccos(x)), \tag{4}$$

where $n \geq 1$ is the number of neural node allocated in the self-adaptive layer. These orthogonal functions can be seen as equivalent to the effect of an activation function. According to the property of an activation function, it must be nonlinear and continuously differentiable. We take into consideration the first derivative of (4) below,

$$\frac{d}{dx} f(x) = \begin{cases} 0 & n = 1, \\ c_1 & n = 2, \\ \displaystyle\sum_{i=0}^{n-1} \frac{c_i i \sin(i \arccos(x))}{\sqrt{1-x^2}} & n \geq 3. \end{cases} \tag{5}$$

It can be found that (4) becomes a linear function when $n = 1, 2$. Thus the number of neurons in this self-adaptive layer should be greater than two. Meanwhile, since the coefficients c_i of all the terms $T_i(x)$ are unique, the weights of self-adaptive layer must be unique as well. Therefore, the problems of local minimum and initial weights do not exist in this layer.

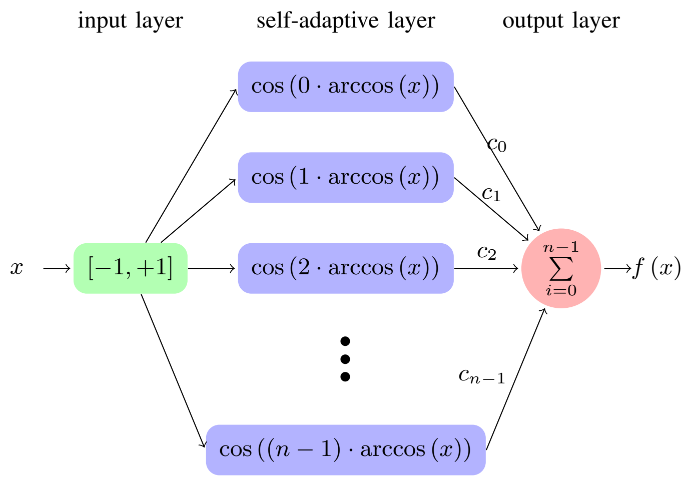
input layer
self-adaptive layer
output layer
x
$[-1,+1]$
$\cos(0 \cdot \arccos(x))$
$\cos(1 \cdot \arccos(x))$
$\cos(2 \cdot \arccos(x))$
$\cos((n-1) \cdot \arccos(x))$
c_0
c_1
c_2
c_{n-1}
$\sum_{i=0}^{n-1}$
$f(x)$

Fig. 2: The self-adaptive layer achieved by Chebyshev polynomial.

Algorithm 1 Module construct of self-adaptive layer.

```
1: function INIT(degree ← $n$)
2:     $c_0 \cdots c_{n-1}$ ← learning parameters allocating    ▷ $n \geq 3$
3: end function
4:
5: function FORWARD($X \leftarrow \{x_0, x_1, \cdots\}$)
6:     for $x \in X$ do
7:         $x' \leftarrow \sum_{i=0}^{n-1} c_i \cdot \cos(i \cdot \arccos(x))$    ▷ (4)
8:     end for
9:     return $X' \leftarrow \{x'_0, x'_1, \cdots\}$
10: end function
```

B. *Implementation*

Since the Chebyshev polynomial function is selected, our proposed layer can be implemented as shown in Fig. 2. Each processing feature only requires the simple calculation of (4) and no complicated activation function is included. In order to achieve our algorithm, we make use of the scientific deep learning framework PyTorch [28] for our implementation.

As discussed in the above section, there are n learning parameters as coefficients that we must allocate for the training algorithm. Then by using the approximation expansion, we can obtain the output feature. This process can be seen as a middle-layer with a unique activation function. The computation can be achieved in the *Module* interface and *forward* function, which are the implementation interface provided by PyTorch. We define the layer structure in Algorithm 1.

As shown in Algorithm 1, the definition of the proposed layer requires the size of n of how many degrees of polynomials we want to expand. The output size is equal to the input feature and each x evolves to x' for the feature enhancement independently, which can be seamlessly connected to the original structure. Because these functions are continuously differentiable, they also accept the gradients of the output and return the gradients of the input. The products passed around

TABLE I: A simple feedforward architecture with two linear layer.

Input Tensor Size	Activation Function	Layer:	$\{parameter = value\}$
$[batch, 1]$	*ReLU*	Linear:	$\{in\_features = 1, out\_features = 10\}$
$[batch, 10]$	None	Linear:	$\{in\_features = 10, out\_features = 1\}$

TABLE II: Added self-adaptive layer with feedforward architecture.

Input Tensor Size	Activation Function	Layer:	$\{parameter = value\}$
$[batch, 1]$	None	Self-Adaptive:	$\{degree = 10\}$
$[batch, 1]$	*ReLU*	Linear:	$\{in\_features = 1, out\_features = 10\}$
$[batch, 10]$	None	Linear:	$\{in\_features = 10, out\_features = 1\}$

are themselves variables, making the evaluation of the graph differentiable. For further detail, the complete Python source code is provided, and can be found at,

`github.com/ChanKaHou/Self-Adaptive-Layer`

III. Experimental Results and Discussion

We applied this self-adaptive layer into various training models for evaluation. All our experiments were conducted on an Nvidia GeForce GTX 1080 with 8.0GB of video memory.

A. Unknown Function Approximation

First, for a simple case, we find the function that can fit a synthetic data set generated by a random perturbation of a variable. A common solution is to apply the feed-forward architecture with several linear layers, and the training parameters are motivated by an activation function in each time step, see Table I and Fig. 3a. In contrast, our proposed self-adaptive layer can achieve the approximation more efficiently and effectively, see Table II and Fig. 3b.

It is worth mentioning that if there are many polynomials used for training, the result could become an unexpected function (see Fig. 3c and 3d). It is because the objective is to make the approximation as close as possible to the unknown function and it must be accomplished by allocating higher degree polynomials. The function can satisfy with all synthetic data so that the loss is zero. However, this extreme result is too perfect that it may raise the overfitting problem in neural networks.

B. Classification Problems

According to the above discussion, we use a plain architecture for the training process as shown in Table III and IV. The model consists of one RNN layer and one linear layer, with an activation function (possible none) following each corresponding layer. We evaluate the classification performance on the same environment with and without using the self-adaptive layer. In order to show the gradient activity, we use the adaptive gradient related optimizer, the Adagrad [29] method, for approximation.

We first analyze the convergence speed of each individual test. Our experiments processed 100,000 iteration steps for

TABLE III: Recurrent Neural Network architecture.

Input Tensor Size	Activation Function	Layer:	$\{parameter = value\}$
$[batch, 28, 28]$	None	RNN:	$\{input\_size = 28, hidden\_size = 64\}$
$[batch, 64]$	*ReLU*	Linear:	$\{in\_features = 64, out\_features = 10\}$

TABLE IV: Added self-adaptive layer with RNN architecture.

Input Tensor Size	Activation Function	Layer:	$\{parameter = value\}$
$[batch, 28, 28]$	None	RNN:	$\{input\_size = 28, hidden\_size = 64\}$
$[batch, 64]$	None	Self-Adaptive:	$\{degree = 10\}$
$[batch, 64]$	*ReLU*	Linear:	$\{in\_features = 64, out\_features = 10\}$

each test. We conducted our experiments four times on both RNN models and took the average of the four to evaluate the performance. We used the same dataset in each test with a batch size of 100 per iteration step.

As shown in Fig. 4, the convergence will speed up with the added self-adaptive layer. The result listed in the figure shows that to reach the same cross entropy, it consumes only 30% to 40% iteration steps with our proposed layer on the MNIST dataset for the RNNs in Table IV. This is a significant improvement. In Fig. 5 the advantages are obvious. It can be seen that the proposed layer can provide a faster learning process, when showing the test accuracy (%) after completing of 200 epochs. The accuracy of self-adaptive layer can achieve to 95.0% requiring 26 epochs compared to the original architecture requiring 71 epochs.

C. Discussion

As a result, we can find that the self-adaptive layer outperforms the original neural network significantly. In all fairness, there are some constraints of our method, listed below.

- We must pay attention to the input data interval. Because a series of approximate polynomials only performs in a particular interval, so out of interval data will issue divergence results.
- There are other orthogonal polynomials that can also work in our method, but the corresponding high degree polynomials are hard to expand. That's why we only present the Chebyshev series implementation in this work.

This method is mainly for approximation of a single-variable function $f(x)$. Approximation theory can also be extended to multi-variable $f(x_0, \cdots, x_{m-1})$ applications, which are possible to apply into signal processing problems. Below is the multi-variable form,

$$f(x_0, \cdots, x_{m-1}) = \sum_{i_0, \cdots, i_{m-1}=0}^{n-1} c_{i_0, \cdots, i_{m-1}} T_{i_0}(x_0) \cdots T_{i_{m-1}}(x_{m-1}),$$

where the coefficients $c_{i_0, \cdots, i_{m-1}}$ are the training parameters. For instance in MNIST, the input feature has a size of 28×28, and the approximation function will result 10 features about the probability of each class. Unfortunately, this algorithm is challenging to implement because it require n^m coefficients

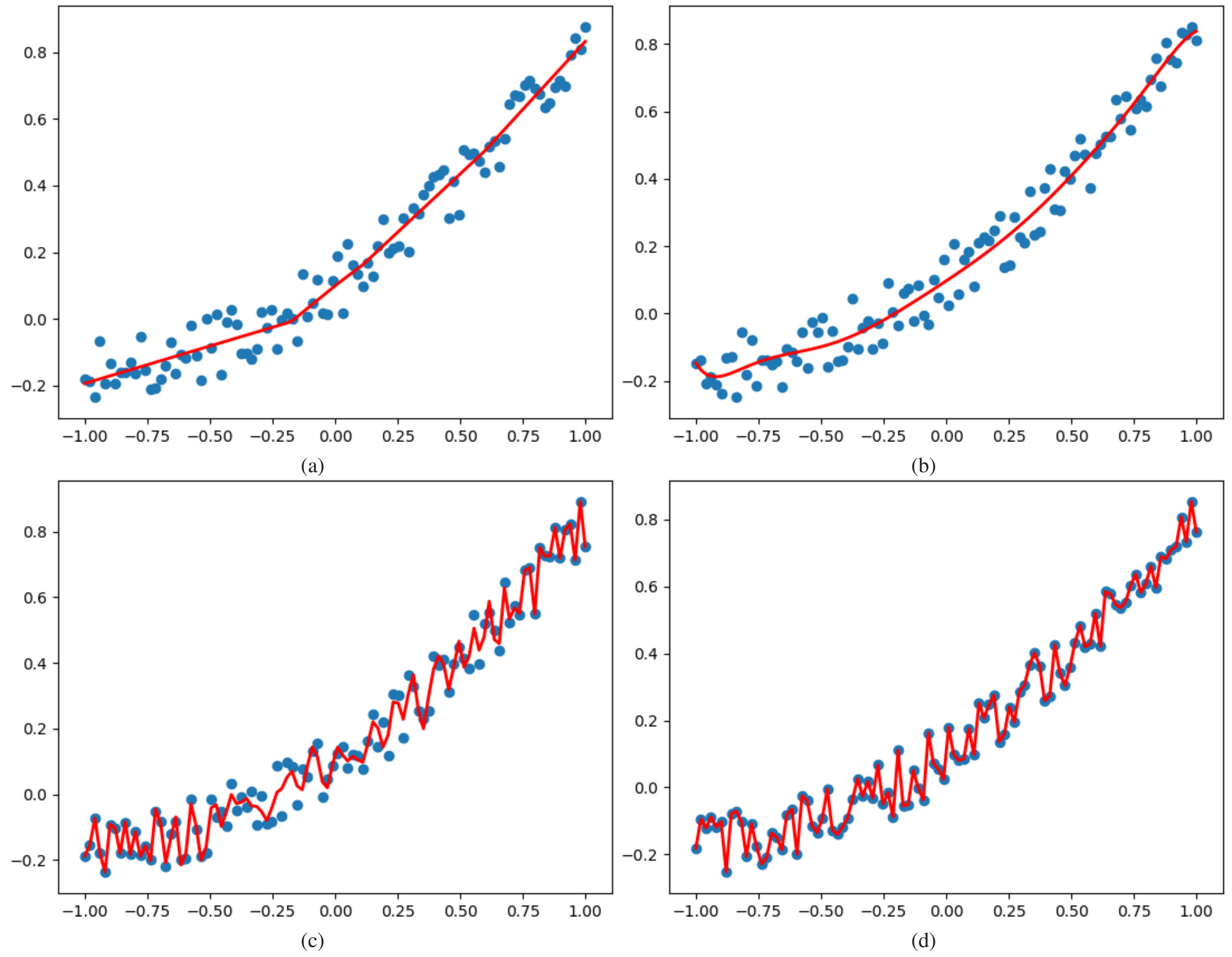

Fig. 3: Unknown Function approximation. (a) present a simple feedforward architecture and (b) to (d) present the self-adaptive layer architecture with different degree ($degree = 10, 100, 200$ respectively) of polynomials.

($n^{28\times28}$ in MNIST) for the training used. This memory requirement is difficult to fulfill, if not impossible, let alone the following time-consuming learning processes.

IV. Conclusion

We present a self-adaptive layer which is based on approximation theory. Our method can be applied to the original architecture as a plugin layer. By introducing the design mapping expansion forms to neural nodes, we are able to prevent the gradient vanishing by allocating n degree polynomials. We show that the choice of n can influence the effectiveness of adapting. To validate the effectiveness, we use models with simple feed-forward and RNN architectures and compare the training processes with and without the added layer. Results show that the model with the added layer can outperform both architectures from start to finish in the convergence speed. The improvement over the discovered model is consistent and significant throughout our experiments. Further work can be done to apply this new layer to other neural networks and with different datasets.

Acknowledgment

This work was supported by the Macao Science and Technology Development Fund through Project 138/2016/A3.

References

[1] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[2] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[3] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[4] J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural computation*, vol. 3, no. 2, pp. 246–257, 1991.

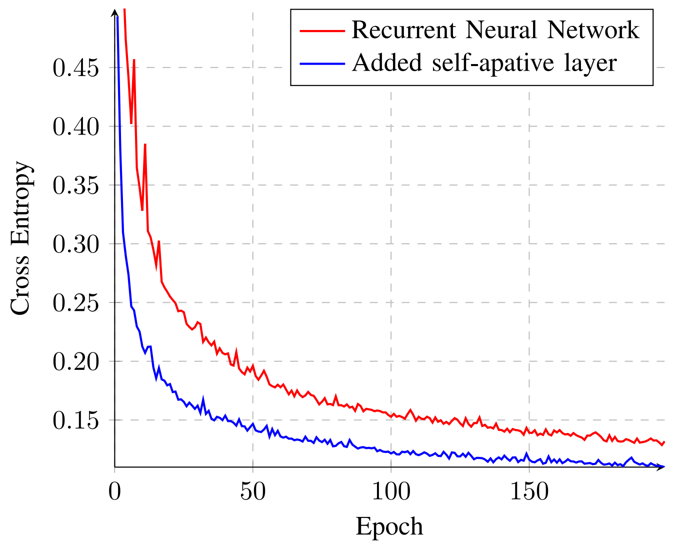

Fig. 4: MNIST, the trend of cross entropy after 200 epochs of training.

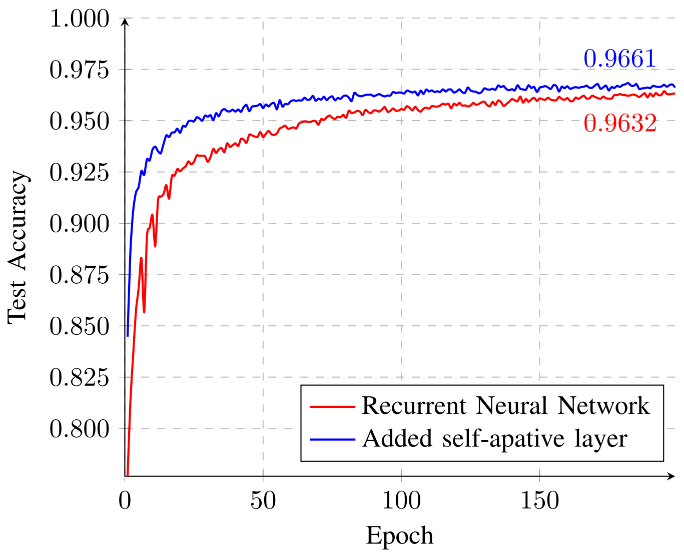

Fig. 5: Test accuracy rates for MNIST after 200 epochs of training.

[5] S.-S. Yang and C.-S. Tseng, "An orthogonal neural network for function approximation," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 5, pp. 779–785, 1996.

[6] P. Werbos, "Neurocontrol and supervised learning: An overview and valuation," *Handbook of intelligent control*, 1992.

[7] T. Lyche, K. Mørken, and E. Quak, "Theory and algorithms for non-uniform spline wavelets," *Multivariate approximation and applications*, vol. 152, 2001.

[8] J. Fan and Q. Yao, *Nonlinear time series: nonparametric and parametric methods*. Springer Science & Business Media, 2008.

[9] S. Qian, Y. Lee, R. Jones, C. Barnes, and K. Lee, "Function approximation with an orthogonal basis net," in *1990 IJCNN International Joint Conference on Neural Networks*. IEEE, 1990, pp. 605–619.

[10] H.-P. Huang and R. Chang, "Unstable backpropagation method in neural networks: A remedy," in *The 2nd International Conference on Automation Technology, Taipei (1992.07)*, 1992.

[11] S. Ferrari and R. F. Stengel, "Smooth function approximation using neural networks," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 24–38, 2005.

[12] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *arXiv preprint arXiv:1312.6120*, 2013.

[13] F. X. X. Yu, A. T. Suresh, K. M. Choromanski, D. N. Holtmann-Rice, and S. Kumar, "Orthogonal random features," in *Advances in Neural Information Processing Systems*, 2016, pp. 1975–1983.

[14] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Advances in neural information processing systems*, 2008, pp. 1177–1184.

[15] K.-H. Chan, S.-K. Im, W. Ke, and N.-L. Lei, "Sinp [n]: A fast convergence activation function for convolutional neural networks," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 365–369.

[16] A. Chernodub and D. Nowicki, "Norm-preserving orthogonal permutation linear unit activation functions (oplu)," *arXiv preprint arXiv:1604.02313*, 2016.

[17] L. Jin, P. Nikiforuk, and M. Gupta, "Direct adaptive output tracking control using multilayered neural networks," in *IEE Proceedings D (Control Theory and Applications)*, vol. 140, no. 6. IET, 1993, pp. 393–398.

[18] M. M. Polycarpou, "Stable adaptive neural control scheme for nonlinear systems," *IEEE Transactions on Automatic control*, vol. 41, no. 3, pp. 447–451, 1996.

[19] Z. Zhou, S. Chen, and Z. Chen, "Fannc: A fast adaptive neural network classifier," *Knowledge and Information Systems*, vol. 2, no. 1, pp. 115–129, 2000.

[20] S. K. Rangarajan, V. V. Phoha, K. S. Balagani, R. R. Selmic, and S. S. Iyengar, "Adaptive neural network clustering of web users," *Computer*, vol. 37, no. 4, pp. 34–40, 2004.

[21] F. Wang, B. Chen, C. Lin, J. Zhang, and X. Meng, "Adaptive neural network finite-time output feedback control of quantized nonlinear systems," *IEEE Transactions on Cybernetics*, vol. 48, no. 6, pp. 1839–1848, 2017.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[25] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[26] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2285–2294.

[27] J. C. Mason and D. C. Handscomb, *Chebyshev polynomials*. Chapman and Hall/CRC, 2002.

[28] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[29] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.