5-2018

Parameterizing and Aggregating Activation Functions in Deep Neural Networks

Luke Benjamin Godfrey
University of Arkansas, Fayetteville

Parameterizing and Aggregating
Activation Functions in Deep Neural Networks

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

by

Luke B. Godfrey
University of Arkansas
Bachelor of Science in Computer Science, 2014
University of Arkansas
Master of Science in Computer Science, 2015

May 2018
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

Dr. Michael S. Gashler
Dissertation Director

Dr. Wing Ning Li
Committee Member

Dr. Xintao Wu
Committee Member

Dr. Giovanni Petris
Committee Member

Abstract

The nonlinear activation functions applied by each neuron in a neural network are essential for making neural networks powerful representational models. If these are omitted, even deep neural networks reduce to simple linear regression due to the fact that a linear combination of linear combinations is still a linear combination. In much of the existing literature on neural networks, just one or two activation functions are selected for the entire network, even though the use of heterogenous activation functions has been shown to produce superior results in some cases. Even less often employed are activation functions that can adapt their nonlinearities as network parameters along with standard weights and biases. This dissertation presents a collection of papers that advance the state of heterogenous and parameterized activation functions. Contributions of this dissertation include

- three novel parametric activation functions and applications of each,
- a study evaluating the utility of the parameters in parametric activation functions,
- an aggregated activation approach to modeling time-series data as an alternative to recurrent neural networks, and
- an improvement upon existing work that aggregates neuron inputs using product instead of sum.

Acknowledgements

I thank Dr. Michael Gashler for his mentorship. He introduced me to the fascinating world of machine learning, a subject about which I have become very passionate under his direction. His time and guidance has been indispensible to my graduate career, and his imagination is truly inspiring. Dr. Gashler made the daunting task of composing a dissertation enjoyable and rewarding, and for that I owe him a great deal of thanks.

I thank my family for all the encouragement, prayer, and love through which they kept me going. I thank my parents for their motivation, support, and example, who often reminded me that God is always in control.

I especially thank my dear wife, Rebekah, who has taken wonderful care of me, staying up late with me on many occasions and being exceptionally patient with her grumpy, sleep-deprived husband. Her love kept me sane and gave me a reason to continue, even when I felt I would never be able to finish this work.

Finally, I thank my great God and Savior, Jesus Christ, without whom none of this would have been possible. The more I study artificial intelligence, the more in awe I am of the one who invented the mind.

Dedication

For my daughter, Lily Birdeen.

Contents

I INTRODUCTION 1

1 Deep Neural Networks 2

2 Activation Functions 4

3 Statement and Summary of Contributions 6

3.1 Dissertation Organization 8

II CONTRIBUTIONS 9

4 A Continuum Among Logarithmic, Linear, and Exponential Functions, and Its Potential to Improve Generalization in Neural Networks 10

4.1 Introduction 10

4.2 Derivation 11

4.3 Analysis 13

4.4 Inner product 14

4.5 Distance 16

4.6 Polynomials 17

4.7 Radial basis function networks 18

4.8 Fourier networks 18

4.9 Proposed architecture 19

4.10 Conclusion 21

5 A Parameterized Activation Function for Learning Fuzzy Logic Operations in Deep Neural Networks 22

5.1 Introduction 22

5.2 Related Work 23

5.3 Approach 25

5.4 Learning Simple Logic Operations 28

5.5 Learning Complex Logic Expressions 30

5.6 Validation 33

5.7 Conclusion 36

6 Neural Decomposition of Time-Series Data 37

6.1 Introduction 37

6.2 Related Work 39

6.2.1 Models for Time-Series Prediction 39

6.2.2 Harmonic Analysis 41

6.2.3 Fourier Neural Networks 44

6.3 High Level Approach . . . 45
6.3.1 Algorithm Description . . . 45
6.3.2 Toy Problem for Justification . . . 49
6.3.3 Toy Problem Analysis . . . 51
6.3.4 Chaotic Series . . . 54
6.4 Implementation Details . . . 56
6.4.1 Topology . . . 57
6.4.2 Weight Initialization . . . 57
6.4.3 Input Preprocessing . . . 58
6.4.4 Regularization . . . 59
6.5 Validation . . . 60
6.6 Conclusion . . . 65

7 Leveraging Product as an Activation Function 68
7.1 Introduction . . . 68
7.2 Related Work . . . 69
7.2.1 Product Unit Neural Networks . . . 69
7.2.2 Gated Units . . . 70
7.2.3 Sum-Product Networks . . . 71
7.3 Approach . . . 71
7.4 Applications . . . 74
7.4.1 Representing Polynomials . . . 74
7.4.2 Generalizing Gated Units . . . 74
7.5 Results . . . 76
7.6 Conclusion . . . 79

8 An Evaluation of Parameterized Activation Functions for Deep Learning 80
8.1 Introduction . . . 80
8.2 Related Work . . . 82
8.2.1 Adaptive Transfer Functions . . . 82
8.2.2 Adaptive Piecewise Linear Units . . . 82
8.2.3 Parametric Rectified Linear Units . . . 83
8.2.4 Parametric Exponential Linear Units . . . 84
8.3 Bendable Linear Units . . . 85
8.4 Experiments . . . 88
8.5 Conclusion . . . 94

III CONCLUSION 95

9 Summary 96

10 Future Work 98

References 99

List of Figures

4.1 A plot of $h(\beta, 3, 7)$. When $\beta = 0$, it correctly calculates $3 + 7 = 10$. When $\beta = 1$, it correctly calculates $3 * 7 = 21$. 13

4.2 A plot of $f(\alpha, x)$ for $\alpha = \{-1, -0.9, -0.8, \cdots, 0.8, 0.9, 1.0\}$ from red to purple. 15

4.3 A plot of $f(\alpha, x)$ for $x = \{-5, -4.5, -4, \cdots, 4, 4.5, 5\}$ from red to purple. . . 15

4.4 A neural network implementation of inner product using soft exponential as an activation function. All of the weights represented with lines in this figure have a value of 1. All other weights have a value of 0. 16

4.5 A neural network implementation of a polynomial, $y = a + bx + cx^2 + dx^3 \ldots$, using soft exponential as an activation function. 17

4.6 A neural network implementation of squared distance using soft exponential as an activation function. To compute Euclidean distance (the square root of this), only one additional network unit would be required. 17

4.7 A plot of the real component of soft exponential over a range of values for α_i. 19

4.8 A plot of the imaginary component of soft exponential over a range of values for α_i. 21

5.1 Equation 5.2 continuously interpolates among three fuzzy logic operations: `nor`, `nxor`, and `and`. By allowing biases (`true` and `false`) and weights (in particular, a weight of -1), this equation can also compute `identity`, `not`, `or`, `xor`, and `nand`. 27

5.2 A comparison of Equations 5.1, 5.2, and 5.3 when computing `true@true`. Equation 5.1 is smoother, but Equation 5.2 is better suited for use with gradient-based optimization. 29

5.3 A comparison of Equations 5.1, 5.2, and 5.3 when computing `true@false`. Only Equation 5.2 is well-suited for use with gradient-based optimization. . . 29

6.1 Three broad classes of models for time-series forecasting: (A) prediction using a sliding window, (B) recurrent models, and (C) regression-based extrapolation. 40

6.2 The predictive model generated by the iDFT for a toy problem with both periodic and nonperiodic components. Blue dots represent training samples, red dots represent testing samples, and the green line represents the iDFT. Two significant problems limit its ability to generalize: (1) The model repeats, ignoring the linear trend, and (2) The extrapolated predictions misalign with the phase of the continuing nonlinear trend. 43

6.3 A diagram of the neural network model used by Neural Decomposition. For each of the k sinusoid units, w_i are frequencies, ϕ_i are phase shifts, and a_i are amplitudes, where $i \in \{1 \dots k\}$. The augmentation function $g(t)$ is shown as a single unit, but it may be composed of one or more units with one or more activation functions. 47

6.4 A comparison of Neural Decomposition with two algorithmic variations showing the importance of certain algorithm details. The data used here is the same data used in Figure 6.2. The full ND model, shown in green, fits very closely to the data that was withheld during training. The cyan curve shows predictions made when the basis functions, including sinusoidal frequencies, were frozen during training. Note that the predictions are out-of-phase, indicating that training these components is essential for effective generalization. The orange curve shows predictions made without including any nonperiodic components among the basis functions, that is, setting the augmentation function $g(t) = 0$. Although the predictions exhibit the correct phase, they fail to fit with the nonperiodic trend. This shows the importance of using heterogeneous basis functions. 48

6.5 (Top) Frequencies of the basis functions of Neural Decomposition over time. (Bottom) Basis weights (amplitudes) over time on the same problem. Note that ND first tunes the frequencies (Top), then finishes adjusting the corresponding amplitudes for those sinusoids (Bottom) (w_A corresponds to ϕ_A and w_B corresponds to ϕ_B). In most cases, the amplitudes are driven to zero to form a sparse representation. After the amplitudes reach zero, the frequencies are no longer modified. 50

6.6 Frequency domain representations of the toy problem (amplitude vs frequency). (Top) Frequencies used by the iDFT. (Bottom) Frequencies used by ND. . . 51

6.7 Neural Decomposition on the Mackey-Glass series. Although it does not capture all the high-frequency fluctuations in the data, our model predicts the location and height of each peak and valley in the series with a high degree of accuracy. 52

6.8 A comparison of the four best predictive models on the monthly unemployment rate in the US. Blue points represent training samples from January 1948 to June 1969 and red points represent testing samples from July 1969 to December 1977. SARIMA, shown in magenta, correctly predicted a rise in unemployment but underestimated its magnitude, and did not predict the shape of the data well. ESN, shown in cyan, predicted a reasonable mean, but did not capture the dynamics of the data. LSTM, shown in orange, predicted the first peak in the data, but leveled off to predict only the mean. Only ND, shown in green, successfully predicted both the depth and approximate shape of the surge in unemployment, followed by another surge in unemployment that followed. 53

6.9 A comparison of the four best predictive models on monthly totals of international airline passengers from January 1949 to December 1960 [16]. Blue points represent the 72 training samples from January 1949 to December 1954 and red points represent the 72 testing samples from January 1955 to December 1960. SARIMA, shown in magenta, learns the trend and general shape of the data. ESN, shown in cyan, predicts a mean but does not capture the dynamics of the actual data. LSTM, shown in orange, predicts a valley and a peak that did not actually occur, followed by a poor estimation of the mean that suggests that it was unable to learn the seasonality of the data. ND, shown in green, learns the trend, shape, and growth better than the other compared models. 55

6.10 A comparison of the four best predictive models on monthly ozone concentration in downtown Los Angeles from January 1955 to August 1967 [73]. Blue points represent the 152 training samples from January 1955 to December 1963 and red points represent the 44 testing samples from January 1964 to August 1967. The compared models include SARIMA, ESN, LSTM, and ND. All four of these models perform well on this problem. Both LSTM (shown in orange) and ESN (shown in cyan) predict with slightly higher accuracy compared to ND. ND, shown in green, has slightly higher accuracy compared to SARIMA (shown in magenta). ARIMA, SVR, and Gashler and Ashmore's model all performed poorly on this problem; rather than include them in this graph, their errors have been reported in Table 6.1 and Table 6.2. 56

6.11 A comparison of two predictive models on a series of oxygen isotope readings in speleothems in India from 1489 AD to 1839 AD [143]. Blue points represent the 250 training samples from July 1489 to April 1744 and red points represent the 132 testing samples from August 1744 to December 1839. Because this time-series is irregularly sampled (the time step between samples is not constant), only SVR and ND could be applied to it. SVR, shown in orange, does not perform well, but predicts a steep drop in value that does not actually occur in the testing data, followed by a flat line. ND, shown in green, performs well, capturing the general shape of the testing samples. . . 60

7.1 Test set misclassifications over time of a WPUNN on the MNIST dataset, fixing $s = 1$ but varying w. The inverse relationship between accuracy and window size demonstrates that WPUNNs are sensitive to the hyperparameter w (window size). Reducing window size solves the major problems associated with training PUNNs, which validates our primary contribution. 73

7.2 A comparison of two neural networks for modeling polynomials. The vertical axis is error (lower is better) and the horizontal axis is polynomial degree. The green curve is the error rate from a WPUNN and the orange curve is the error rate from a ReLU network. As expected, the WPUNN consistently yields a lower error rate than the ReLU model. The noticable increase in loss in the WPUNN for $d = 9$ and $d = 10$ can be attributed to the depth of the network being less than $log(d)$. 75

7.3 Forecasts made by two models on a time-series of Mauna Loa CO2 readings. Blue points represent training data, red points represent withheld testing data, the green curve represents the forecast by a WPUNN, and the orange curve represents the forecast by a LSTM network. Although the LSTM network yields a more accurate prediction, the WPUNN model is faster and uses a simpler architecture. 77

8.1 A visualization of Bendable Linear Units, varying α, with β fixed at 0.9. The closer α is to 0, the more it is like LReLU [115], and the closer α is to 1, the more it is like SoftPlus [51]. 85

8.2 A visualization of Bendable Linear Units, with α fixed at 0.5. The closer β is to 0, the more it is like the identity function, and the closer β is to 1, the steeper the bend. 86

8.3 One of the β values from the lowest layer of the WRN-16-4 network using the BLU-β activation. The initial value, which is about 0.2, is selected from a uniform random distribution. During the course of training, the network tunes this value to almost 0, forming a kind of learned residual connection. . 92

8.4 Selected training curves of test set accuaracy on the CIFAR-10 task from our first experiment. We include curves for BLU, ELU (in orange), PELU (in red), ReLU (in green), and PReLU (in yellow). BLU learns the fastest, but does not converge to as good a final accuracy as PELU and ELU. PELU and ELU achieve approximately the same final result, but PELU converges faster. PReLU, on the other hand, yields significantly better results than ReLU. Thus, for both PELU and PReLU, we observe a benefit to the parameterization. 93

List of Tables

5.1 Average errors of a deep neural network (DNN), an ensemble of ANFIS fuzzy classifiers (EFC), our model, and the expressions obtained from “snapping” the weights in our network (Snapped) on the validation data for five classification problems. Best results are **bolded**. 33

5.2 Logical expressions learned by our model for three of the datasets. These expressions were formed by “snapping” all parameters of the network to the nearest whole value. Expressions for the other two datasets were omitted for the sake of brevity. 35

6.1 Mean absolute percent error (MAPE) on the validation problems for ARIMA, SARIMA, SVR, Gashler and Ashmore, ESN, LSTM, and ND. Best result (smallest error) for each problem is shown in **bold**. 65

6.2 Root mean square error (RMSE) on the validation problems for ARIMA, SARIMA, SVR, Gashler and Ashmore, ESN, LSTM, and ND. Best result (smallest error) for each problem is shown in **bold**. 66

8.1 The wide residual network (WRN) topology [172] used in some of our experiments. A WRN has two hyperparameters: d (which controls depth) and k (which controls width). n, used in the table, is defined as $\frac{d-4}{6}$. The final column, Number, is how many copies of the layer are used in succession. . . 90

8.2 The final CIFAR-10 test set accuracy for each activation/topology pair tested in our first experiment. The best activation for each topology is shown in **bold**. Parametric activations tend to achieve higher accuracy than their non-parametric counterparts, although not always by a significant margin. 91

8.3 The final CIFAR-10 test set accuracy for each activation using a WRN-16-4 topology in our second experiment, limiting training time to 100 epochs. The parametric activations learn faster than the non-parametric activations. The best result is shown in **bold**. 92

8.4 The final CIFAR-10 test set accuracy for each activation tested in our third experiment (using WRN-40-4 with no residual connections). The difference in accuracy compared with WRN-40-4 *with* residual connections is reported in the last column. BLU performed the best in this experiment, both in terms of accuracy and difference in loss. PELU, marked with “x”, diverged during training for several random seeds. ELU achieved good results, despite not being able to approximate the identity function. Though not as good as ELU or BLU, PReLU vastly outperforms ReLU. The best result (highest accuracy and lowest difference in accuracy) is shown in **bold**. 94

Part I

INTRODUCTION

Chapter 1

Deep Neural Networks

An artificial neural network is a machine learning model that simulates, to the best of our knowledge, how the brain processes electrical signals. In this abstraction, neural networks are made up of neurons (also called nodes) and synapses (also called connections) that link neurons together. A neuron's output is computed by aggregating its inputs (i.e. as a weighted sum) and activating the result by applying an activation function such as `tanh`. One widely used class of neural networks is multi-layer perceptrons, in which neurons are arranged in layers, with the outputs of one layer being used as inputs to the next. It has been shown that such a neural network with only a single hidden layer (that is, one layer between the input and the output) can serve as a universal function approximator [76]. Neural networks with many hidden layers are called deep neural networks, and depths can range from only six layers [26] to more than 100 [78].

There is a vast body of work involving deep neural networks. Many optimization techniques have been studied [70, 157, 95], much work has considered various topologies [118, 148], and several general approaches have been proposed, such as recurrent LSTM networks [75, 49], convolutional networks [102], and reservoir networks [81]. Activation functions, which provide neural networks with the crucial nonlinearities necessary to generalize well, have also been well-studied [33, 93]. Some of the most popular activation functions are `tanh` and rectified linear.

Neural networks are powerful adaptive models with applications in many disciplines. With the backpropagation algorithm, neural networks can be trained in a straightforward manner by gradient-based optimization techniques like stochastic gradient descent and RMSProp [157]. Several variations of these models are particularly good at certain tasks, such as convolutional neural networks for image classification [102] and LSTM for time-series analysis [75]. In every neural network, the topology, connections, and nonlinearities play a crucial role to learning.

Chapter 2

Activation Functions

The nonlinear activation functions applied by each neuron in a neural network are critically important for effective learning. In fact, if these are omitted, even deep neural networks reduce to simple linear regression. This is due to the fact that a linear combination of linear combinations is still a linear combination. Thus, activation functions are essential for making neural networks powerful representational models.

For years, sigmoidal activation functions such as `tanh` have been popular choices [91]. More recently, rectified linear units (ReLUs) have been shown to possess desirable properties [125, 174], as well as a number of variations including leaky ReLUs [115], randomized ReLUs [165], parametric ReLUs [67], and more. Other widely used activations include exponential linear units (ELUs) [27] and scaled ELUs (SELUs) [96].

While these functions perform well empirically, little theoretical basis has been found to justify their extensive use over many other potential functions. In much of the existing literature on neural networks, just one or two activation functions are selected for the entire network, even though the use of heterogenous activation functions has been shown to produce superior results in some cases [159]. Indeed, although using a different activation function for each layer is common, little work has explored mixing multiple activation functions side-by-side within each layer.

Even less often employed are activation functions that can adapt their nonlinearities as network parameters along with standard weights and biases. An adaptable activation function similar to rectified linear but with an arbitrary number of bends was recently proposed and found to achieve state-of-the-art performance on the CIFAR-10 and CIFAR-100 datasets over standard rectified linear networks at the time of its proposal [2]. Parameterized activation functions are not yet widely used, but interest in the topic appears to be building as more research reveals their effectiveness [134, 136].

Chapter 3

Statement and Summary of Contributions

The aggregation of heterogenous activation functions and the parameterization of activation functions can enable neural networks to model data more quickly and more accurately. Heterogenous activation functions are well-suited for modeling data sampled from the composition of heterogenous signals, while parametric activation functions can be used to promote simpler models to reduce overfitting and generalize more effectively.

Contributions of this dissertation include

- three novel parametric activation functions and applications of each:
  - Soft Exponential (SoftExp), which interpolates between logarithm, linear, and exponential functions,
  - the first adaptive transfer function that learns fuzzy logic operations by gradient descent, and
  - Bendable Linear Unit (BLU), which synthesizes properties from parametric rectified linear units (PReLUs), exponential linear units (ELUs), and scaled exponential linear units (SELUs) and enables implicit residual connections in deep networks,
- a study evaluating the utility of the parameters in parametric activation functions, in which we find the following:
  - parametric activations achieve (marginally) higher accuracy than their non-parametric counterparts,
  - parametric activations tend to converge more quickly than non-parametric activations, and

  - parametric activations that can approximate the identity function are more robust than non-parametric activations when removing residual connections,

- an aggregated activation approach to modeling time-series data as an alternative to recurrent neural networks, which

  - empirically shows why the Fourier transform provides a poor initialization point for generalization and how neural network weights must be tuned to properly decompose a signal into its constituent parts,

  - demonstrates the necessity of an augmentation function in Fourier and Fourier-like neural networks and shows that components must be adjustable during the training process, observing the relationships between weight initialization, input preprocessing, and regularization in this context, and

  - unifies these insights to describe a method for time-series forecasting and demonstrates that this method is effective at generalizing for some real-world datasets,

- an improvement upon existing work that aggregates neuron inputs using product instead of sum, demonstating that

  - gradient-based optimization can be used effectively with windowed product units,

  - windowed product is as effective as traditional nonlinearities like rectified linear units (ReLU),

  - windowed product unit neural networks can generalize gated units in recurrent neural networks, and

  - our method solves the major problems associated with training product unit neural networks.

3.1 Dissertation Organization

Chapters 4 through 6 of this dissertation consist of three works that have been published as a result of this dissertation, and Chapters 7 through 8 consist of two works that are currently under consideration for publication. The references for the three published works are listed here, and are ordered according to the chapter in which they appear in this dissertation:

- Luke B Godfrey and Michael S Gashler. A continuum among logarithmic, linear, and exponential functions, and its potential to improve generalization in neural networks. In *Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K), 2015 7th International Joint Conference on*, volume 1, pages 481–486. IEEE, 2015

- Luke B Godfrey and Michael S Gashler. A parameterized activation function for learning fuzzy logic operations in deep neural networks. In *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*. IEEE, 2017

- Luke B Godfrey and Michael S Gashler. Neural decomposition of time-series data for effective generalization. *IEEE transactions on neural networks and learning systems*, 2017

It should also be noted that the publication in Chapter 6 builds upon the author's Masters' thesis (2015), but is made up of substantially new content not included in that thesis.

Part II

CONTRIBUTIONS

Chapter 4

A Continuum Among Logarithmic, Linear, and Exponential Functions, and Its Potential to Improve Generalization in Neural Networks

Abstract: We present the soft exponential activation function for artificial neural networks that continuously interpolates between logarithmic, linear, and exponential functions. This activation function is simple, differentiable, and parameterized so that it can be trained as the rest of the network is trained. We hypothesize that soft exponential has the potential to improve neural network learning, as it can exactly calculate many natural operations that typical neural networks can only approximate, including addition, multiplication, inner product, distance, polynomials, and sinusoids.

4.1 Introduction

Each neuron in an artificial neural network applies a non-linear activation function to a weighted sum of its inputs. The activation function serves the important role of enabling the neural network to fit to non-linear curves and surfaces. If omitted, even deep multi-layered neural networks reduce to be functionally equivalent to simple linear regression. Hence, the activation function endows the neural network with its representational power.

One might ask, *which activation function is best for neural networks?* For years, the logistic and tanh functions have been popular choices [91]. More recently, rectified linear units have been shown to possess desirable properties [125, 174]. While these functions perform well empirically, little theoretical basis has been found to justify their extensive use over many other potential functions. We present the soft exponential function, a novel activation function with many desirable theoretical properties. It continuously interpolates between logarithmic, linear, and exponential activation functions. It enables neural networks to exactly compute many natural mathematical structures that can only be approximated by neural networks that use traditional activation functions, including addition, multiplication, exponentiation, dot product, Euclidean and L-norm distance, polynomials, Gaussian radial basis functions, and Fourier neural networks.

The next section derives soft exponential and the remainder of the chapter discusses its desirable properties.

4.2 Derivation

It is well known that multiplication can be implemented by means of addition in logarithmic space. That is,

$$p * q = e^{(\log_e p)+(\log_e q)}. \tag{4.1}$$

This property can enable neural networks that use a mixture of logarithmic, linear, and exponential activation functions to exactly perform the basic mathematical operation of multiplication. However, using a mixture of different activation functions in a single neural network adds a significant component of complexity. Specifically, it leaves the user to determine which activation function should be used with each neuron in the network. If a function can be found that continuously generalizes between logarithmic, linear, and exponential functions, then a neural network with a single activation function would be empowered to autonomously learn to add, multiply, exponentiate, and compute the logarithms as needed to accomplish arbitrary tasks. Because these mathematical operations have proven to have significant value in nearly all other areas of science, it is natural to suppose that neural networks should be given the ability to perform the same operations when they attempt to autonomously model various phenomena.

A simple equation that continuously interpolates between linear and exponential functions is

$$g(\alpha, x) = \frac{e^{\alpha x} - 1}{\alpha} + \alpha. \tag{4.2}$$

Note that $\lim_{\alpha \to 0} g(\alpha, x) = x$, and $g(1, x) = e^x$. This function does not become a logarithmic function (i.e. when $\alpha = -1$), so it does not provide a complete solution to our objective. However, we can invert g with respect to x to obtain a function that interpolates between logarithmic and linear functions:

$$g^{-1}(\alpha, x) = \frac{\log_e(1 + \alpha(x - \alpha))}{\alpha}. \tag{4.3}$$

Since g and g^{-1} are equivalent when $\alpha = 0$, we can mathematically piece them together along that edge without breaking continuity. We negate α in the case of the inverse function and obtain the following continuous piecewise function:

$$f(\alpha, x) = \begin{cases} -\frac{\log_e(1 - \alpha(x + \alpha))}{\alpha} & \text{for} \quad \alpha < 0 \\ x & \text{for} \quad \alpha = 0 \\ \frac{e^{\alpha x} - 1}{\alpha} + \alpha & \text{for} \quad \alpha > 0. \end{cases} \tag{4.4}$$

Equation 4.4 interpolates between logarithmic, linear, and exponential functions. Although it is spliced together, it is continuous both with respect to α and with respect to x, and has a number of properties that render it particularly useful as a neural network activation function. We call f the soft exponential activation function.

We can now address the challenge of creating a continuum of operations between addition and multiplication. By substituting f into Equation 4.1, we obtain a continuous generalization between these two operations:

$$h(\beta, p, q) = f\left(\beta, f(-\beta, p) + f(-\beta, q)\right). \tag{4.5}$$

If $\beta = 0$, this function adds p and q. If $\beta = 1$, it multiplies p and q. Figure 4.1 illustrates this continuum between addition and multiplication with the arbitrary values $p = 3$ and

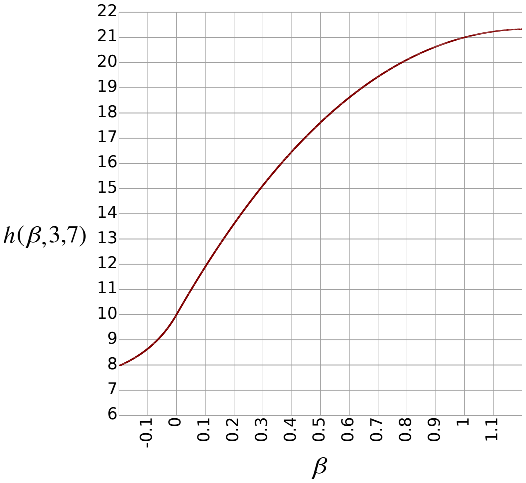

Figure 4.1: A plot of $h(\beta, 3, 7)$. When $\beta = 0$, it correctly calculates $3+7 = 10$. When $\beta = 1$, it correctly calculates $3 * 7 = 21$.

$q = 7$. At $\beta = 0$, it correctly calculates $3 + 7 = 10$, and at $\beta = 1$, it correctly calculates $3 * 7 = 21$.

4.3 Analysis

Some of the nice properties of soft exponential include:

- $f(-1, x) = \log_e(x)$
- $f(0, x) = x$
- $f(1, x) = e^x$
- For other values of α, $f(\alpha, x)$ does something continuous and reasonable.
- The equation is simple, and can be implemented in code with very few operations.
- It appears reasonably smooth when plotted. (See Figures 4.2 and 4.3.)
- Negating α inverts the function, such that $f^{-1}(\alpha, x) = f(-\alpha, x)$.

- For any constant value of α, $f(\alpha, x)$ is monotonic.

- It is continuously differentiable with respect to x,

$$\frac{\partial f}{\partial x} = \begin{cases} \frac{1}{1-\alpha(\alpha+x)} & \text{for} \quad \alpha < 0 \\ e^{\alpha x} & \text{for} \quad \alpha \geq 0 \end{cases} \tag{4.6}$$

  because

$$\lim_{\alpha \to 0^+} \frac{\partial f}{\partial x} \equiv \lim_{\alpha \to 0^-} \frac{\partial f}{\partial x} \equiv 1.$$

- And it is continuously differentiable with respect to α,

$$\frac{\partial f}{\partial \alpha} = \begin{cases} \frac{\log_e(1-(\alpha^2+\alpha x)) - \frac{2\alpha^2+\alpha x}{\alpha^2+\alpha x-1}}{\alpha^2} & \text{for} \quad \alpha < 0 \\ \frac{x^2}{2} + 1 & \text{for} \quad \alpha = 0 \\ \frac{\alpha^2+(\alpha x-1)e^{\alpha x}+1}{\alpha^2} & \text{for} \quad \alpha > 0 \end{cases} \tag{4.7}$$

  because

$$\lim_{\alpha \to 0^+} \frac{\partial f}{\partial \alpha} \equiv \lim_{\alpha \to 0^-} \frac{\partial f}{\partial \alpha} \equiv \frac{x^2}{2} + 1.$$

- Because it is differentiable, it is possible to train a neural network with soft exponential using gradient descent. The alpha parameter of the activation function is updated in the same manner as the weights, by stepping in the gradient direction that reduces some objective function.

4.4 Inner product

One operation we might want to generalize is inner product. The inner product is typically implemented as, $\mathbf{p} \cdot \mathbf{q} = p_0 q_0 + p_1 q_1 + p_2 q_2 + \ldots$. Inner product could be implemented using a 3-layer neural network as depicted in Figure 4.4. This network uses soft exponential for the activation function in each of its units. The first layer computes the logarithm of all the elements in $\mathbf{p}$ and $\mathbf{q}$. (All the units in this layer use $\alpha = -1$.) The second layer adds

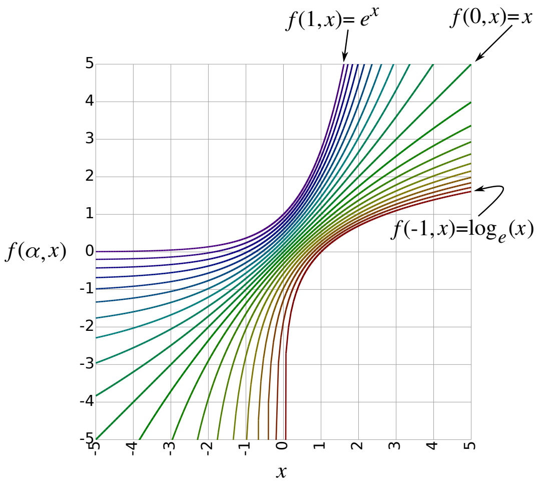

Figure 4.2: A plot of $f(\alpha, x)$ for $\alpha = \{-1, -0.9, -0.8, \cdots, 0.8, 0.9, 1.0\}$ from red to purple.

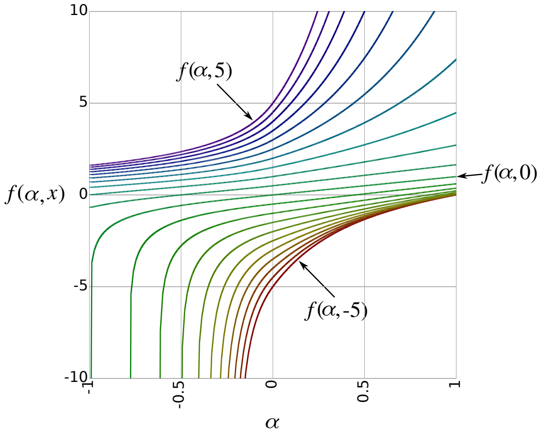

Figure 4.3: A plot of $f(\alpha, x)$ for $x = \{-5, -4.5, -4, \cdots, 4, 4.5, 5\}$ from red to purple.

corresponding elements of **p** and **q**, and exponentiates the result. (All the units in this layer use $\alpha = 1$.) The third layer sums all the pair-wise products together. (The unit in this layer uses $\alpha = 0$.)

One possible use for this generalization of inner product is to implement a neural network version of matrix factorization, a useful algorithm for recommender systems [98] and missing value imputation for sparse matrix completion [12]. Matrix factorization has also proved to be effective for document clustering [166], text mining and spectral data analysis [7], and molecular pattern discovery [11]. A neural network with our activation function can exactly

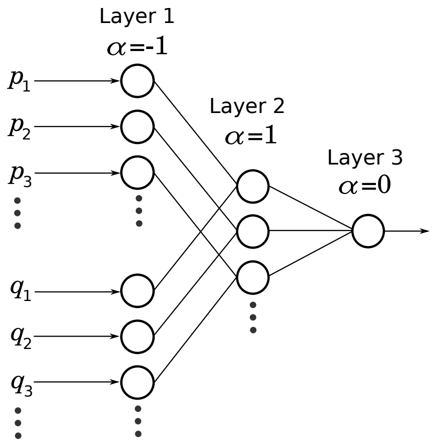

Figure 4.4: A neural network implementation of inner product using soft exponential as an activation function. All of the weights represented with lines in this figure have a value of 1. All other weights have a value of 0.

compute inner product and matrix factorization, and thus it should be able to achieve accuracy at least as good as approaches that do not use neural networks. Because of the flexibility of this generalized approach, it has the potential to outperform direct matrix factorization. For example, in a recommender system, our approach facilitates augmenting user and item profile vectors with static profile vectors for addressing the cold-start problem [98].

4.5 Distance

Suppose we want to compute the distance between two vectors, $\mathbf{p}$ and $\mathbf{q}$. This could also be done with a neural network that uses soft exponential for its activation functions. To do this, we will use the property,

$$a^b = e^{b \log_e(a)}.$$

Figure 4.6 shows a neural network that computes the squared distance between two vectors. (If you want to take the square root, to make it Euclidean distance, just change the unit in layer 3 to use $\alpha = -1$, and add a layer 4 with one unit. This unit would use $\alpha = 1$, and its incoming weight would be set to 0.5.)

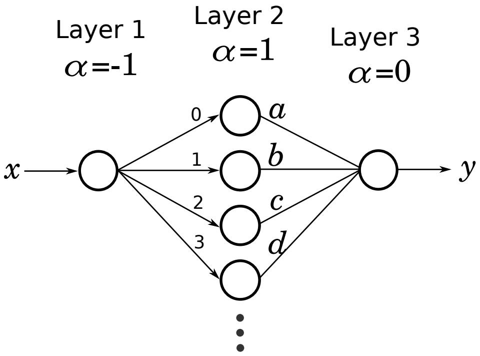

Figure 4.5: A neural network implementation of a polynomial, $y = a + bx + cx^2 + dx^3 \ldots$, using soft exponential as an activation function.

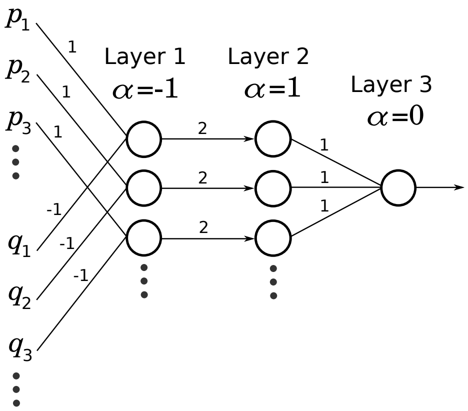

Figure 4.6: A neural network implementation of squared distance using soft exponential as an activation function. To compute Euclidean distance (the square root of this), only one additional network unit would be required.

4.6 Polynomials

Figure 4.5 shows a neural network that exactly computes an arbitrary polynomial. Multivariate polynomials could also be implemented by simply adding additional units on the input end.

4.7 Radial basis function networks

A gaussian radial basis kernel uses the formula,

$$e^{-rs},$$

where r is a weight that controls the squared radius of the kernel, and s is either the squared distance between the input vector and the center of the kernel, or the inner product with the input vector. This function is important to a number of classification models, including support vector machines that use a radial basis function and radial basis function networks [138, 18, 130]. This could be implemented in a network using only f as an activation function by simply adding a single unit with $\alpha = 1$ to the neural networks in Figures 4.4 or 4.6. The weight feeding into this unit would be $-r$. If we added a layer to combine several of these, we would have a radial basis function network without using any specialized units.

Although it is already well-known that neural networks are universal function approximators [29], it is worth noting that soft exponential enables common architectures to be exactly implemented using a neural network with minimal architectural overhead. If a simple model sufficiently models a set of data, it is generally preferable and yields better predictions than an unnecessarily complex one. If these architectures were implemented using a network with a sigmoidal activation function, for example, the resulting models would be very large networks that would probably take more training data to train it to generalize well.

4.8 Fourier networks

Fourier neural networks use a sinusoidal activation function to transform a signal from the time or space domain to the frequency domain in a process similar to the Fourier transform [141, 154, 182]. If α is allowed to have a complex value, soft exponential can be used as the activation function in a Fourier neural network. Let α_r be the real component of α, and α_i be the imaginary component of α, such that $\alpha = \alpha_r + i\alpha_i$. For simplicity, we assume that x

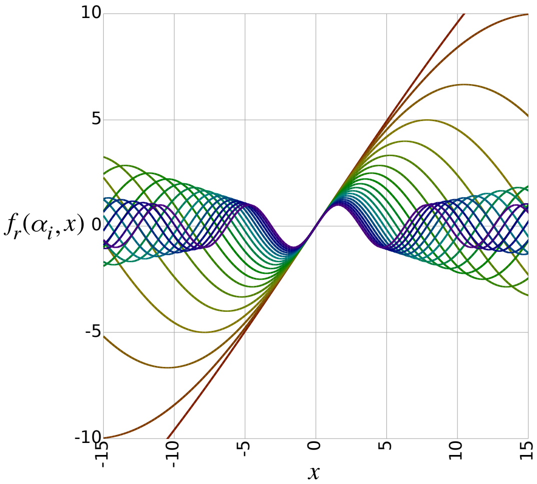

Figure 4.7: A plot of the real component of soft exponential over a range of values for α_i.

is real, and $\alpha_r = 0$. Then the equation for f becomes

$$f(\alpha_i, x) = \frac{\sin(\alpha_i x)}{\alpha_i} + i\left(\alpha_i - \frac{\cos(\alpha_i x) + 1}{\alpha_i}\right). \tag{4.8}$$

Without these assumptions, the resulting equation contains several additional terms. Figures 4.7 and 4.8 show the real and imaginary components respectively of f over a range of values for α_i. It can be seen in these figures that the imaginary component of α determines the frequency of the sinusoidal wave. (Although it also affects the amplitude, this is not significant because the outgoing weight can compensate to achieve any desired amplitude).

We have shown that Fourier networks are effective for extrapolating real-world time-series data [44]. Because soft exponential can be logarithmic, exponential, linear, or sinusoidal when α is allowed to be complex, we can create a Fourier network with only this activation function and achieve the same level of accuracy for generalization and extrapolation.

4.9 Proposed architecture

We conclude our discussion by describing a deep neural network architecture that could potentially use this novel activation function to autonomously achieve all of these representational capabilities as needed to address a wide range of challenges. Because complex values for α cause each unit to output two values, instead of one, it may not be immediately clear how to apply such a network to arbitrary problems. However, if the α parameter values in the output layer are constrained to take only real values, then this network will behave like traditional neural networks, mapping from any number of input values to any number of output values. Allowing hidden units to take on complex values for α should not present any problems because the additional values may simply be fed into the next layer as if the preceding layer were twice as big. Hence it should be reasonable to use f as the activation function for every unit in a deep neural network.

The α parameter for each unit could be initialized to $0 + 0i$. This has the very desirable property of initially causing the entire network to behave like linear regression. As training proceeds, it will take on non-linearities only as necessary to fit the data. All of the weights would be initialized with random values drawn from a normal distribution, then normalized such that the primary eigenvalue is 1. Since all of the activation functions are initially the identity function, the problem of vanishing gradients is initially mitigated, enabling very deep networks to be trained efficiently. This activation function does not impose any particular topology on the rest of the network, so the layers could fully-connected or arranged with sparse connections, such as in convolutional layers.

Likewise, the differentiability of soft exponential facilitates optimization with batch gradient descent, stochastic gradient descent, or many other optimization techniques. α can be updated along with the weights in the manner of steepest descent. L^1 regularization should be applied to promote sparsity. It can be observed that the various common architectures that we can demonstrated with this activation function use sparse connections. It follows, therefore, that L^1 regularization may be expected to work particularly well with this activation function. Note that L^1 regularization can be applied to the α parameter as well as the weights of the network. When α is pulled toward zero, the network approaches linear regression. Hence, regularizing the α parameter has the desirable effect of causing the surface

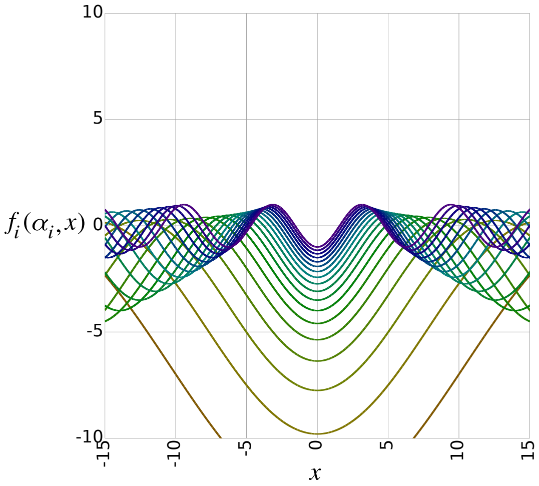

Figure 4.8: A plot of the imaginary component of soft exponential over a range of values for α_i.

represented by the neural network to straighten out.

4.10 Conclusion

We presented a novel activation function, soft exponential, that continuously generalizes among logarithmic, linear, and exponential functions. This function exhibits many desirable theoretical properties that make it well-suited for use as an activation function with neural networks. Empirical validation of these theoretical properties still needs to be performed as future work. Because of the significant potential that this activation function has to impact the effectiveness of deep neural networks, we are anxious to share these ideas with the broader research community now, instead of waiting for our attempts at achieving validation, so that the community may participate in the process of discovering its potential and limitations.

Chapter 5

A Parameterized Activation Function for Learning Fuzzy Logic Operations in Deep Neural Networks

Abstract: We present a deep learning architecture for learning fuzzy logic expressions. Our model uses an innovative, parameterized, differentiable activation function that can learn a number of logical operations by gradient descent. This activation function allows a neural network to determine the relationships between its input variables and provides insight into the logical significance of learned network parameters. We provide a theoretical basis for this parameterization and demonstrate its effectiveness and utility by successfully applying our model to five classification problems from the UCI Machine Learning Repository.

5.1 Introduction

Neural networks are powerful adaptive models with applications in many disciplines. With the backpropagation algorithm, neural networks can be trained in a straightforward manner by gradient-based optimization techniques like stochastic gradient descent and RMSProp [157]. Some of these models, such as convolutional neural networks, learn parameters that can be visualized, interpreted, and understood in some cases [173]. Most neural networks, however, are considered black boxes [92, 144] and it is difficult to determine the semantic meanings behind learned weights.

Fuzzy inference systems, built on fuzzy logic, are also powerful models. Unlike neural networks, fuzzy inference systems are straightforward to interpret and often use linguistic values [170]. In fact, these systems are functionally equivalent to a subset of neural networks [84]. Fuzzy inference systems are less general than neural networks, however, and many neural network techniques are not easily translated into the domain of fuzzy logic.

For these reasons, there is great interest in combining neural networks with fuzzy logic and fuzzy inference systems. Adaptive fuzzy systems have been studied for decades [85, 83, 111], and neural fuzzy modeling continues to be an active topic of research [20, 17, 94]. One purpose of combining these techniques is to produce a model with the flexibility and accuracy of black-box neural networks and the interpretability of fuzzy systems.

Although neural fuzzy systems are well-studied, existing approaches primarily focus on combining specific logical operations in a predefined manner, the most common being an `or` of `ands` [85, 83, 104]. Models that restrict themselves to particular kinds of expressions are limited in the insights they can offer about given datasets. A system that can adaptively choose from a larger set of logical operations, on the other hand, would be able to provide us with more knowledge about the relationships between its various inputs.

We present a deep learning architecture for learning fuzzy logic expressions by using a novel adaptive transfer function. Our model uses an innovative, parameterized, differentiable activation function that can learn a number of logical operations by gradient descent. Parameters learned by our model can be interpreted as fuzzy rules and combined to form complex logic expressions, allowing a glimpse into the knowledge gleaned during the training process. In Section 5.6, we report the results of applying our model to five classification problems taken from the UCI Machine Learning Repository [5]. We find that our model is able to learn complex logical expressions and to achieve accuracy comparable to a standard deep neural network with `tanh` activation functions.

5.2 Related Work

Fuzzy logic [97] extends boolean logic into a continuous domain. Typically, `false` is represented as `0`, `true` is represented as `1`, and values in between indicate a corresponding "fuzzy" degree of uncertainty. A typical set of fuzzy operators that generalize the behavior of boolean logic are:

$$
\begin{aligned}
\texttt{identity}(x) &= x \\
\texttt{not}(x) &= 1 - x \\
\texttt{or}(x, y) &= 1 - (1 - x) \cdot (1 - y) \\
\texttt{xor}(x, y) &= x + y - 2 \cdot x \cdot y \\
\texttt{and}(x, y) &= x \cdot y \\
\texttt{nor}(x, y) &= (1 - x) \cdot (1 - y) \\
\texttt{nxor}(x, y) &= 1 - (x + y - 2 \cdot x \cdot y) \\
\texttt{nand}(x, y) &= 1 - x \cdot y
\end{aligned}
$$

One of the most common applications of fuzzy logic is to control systems [105] using a fuzzy inference system [171]. A typical fuzzy logic controller uses Gaussian membership functions to "fuzzify" inputs (which may be linguistic [170]), a set of (fuzzy) logical IF-THEN rules to apply to the fuzzified inputs, and a function that aggregates and "defuzzifies" the result to a crisp value that determines the system's action [84]. For example, a fuzzy inference system for an autonomous car might have this inference rule: *IF speed limit IS low OR traffic IS dense THEN speed = slow*. In this example, *low*, *dense*, and *slow* are linguistic values that correspond to functions that map raw sensor inputs to real numbers in the range [0..1] where 0 indicates definitely not in the given set, 1 indicates definitely in the set, and anything else represents how typical the input is for the given set. 100 km/h might be 0.5 in the moderate speed set and 0.8 in the fast speed set.

The combination of fuzzy logic with neural networks has been termed "fuzzy modeling" [85], "neural fuzzy systems" [111], and "adaptive fuzzy systems" [83]. These systems were extensively studied in the 1990s [99, 21, 65, 14, 181, 86], and one of the primary reasons was that the weights and parameters of a neural fuzzy system could be interpreted more meaningfully than in a traditional neural network [28]. More recently, fuzzy neural networks have been applied to tracking control [17] and other nonlinear dynamics [94].

One kind of neural fuzzy system is any neural network that directly models a fuzzy inference system [109, 110, 21]. These models generally have five layers: 1) an input layer,

2) a membership layer to fuzzify input, 3) a rules layer that computes products of the second layers outputs, 4) a normalization layer to defuzzify signals, and 5) a summation layer to produce the output. Inputs and outputs to this kind of system are crisp, and the fuzzy logic takes place in the hidden layers of the network. In 1993, Jang and Sun showed that these neural networks are functionally equivalent to fuzzy inference systems [84]. The learning that occurs in this kind of neural fuzzy system tunes the member functions, adjusting means and standard deviations, in addition to the combination weights in the output layer. The rules layer in these models is implemented as a product-of-inputs [85], which is a logical `and`. The summation layer performs the logical `or`, and so most of these models result in a logical `or` of `ands` (or a `max` of `mins` [104]).

Another type of neural fuzzy system is any system that uses both fuzzy logic and neural networks as parts of a whole. Chen et. al recently applied this kind of model to solar radiation forecasting, in which the authors used a fuzzy inference system to combine the predictions three separate neural networks like a weighted ensemble [20]. Other models combine fuzzy systems with genetic algorithms [160] or with a Kalman filter [85]. Still others allow inputs, weights, and outputs to be fuzzy [103]. Kwan and Cai proposed the use of "fuzzy neurons" that combine an aggregation function and an activation function with some number of membership functions [104].

5.3 Approach

We take an approach similar to the common five-layer neural fuzzy system [83], although we use a deeper network. Our model is unique not in its topology in the interpretability of its weights, however, but in the adaptive activation function we use that is able to learn several logical operations. Our activation function is parameterized, continuous, and differentiable, and can therefore be tuned by gradient descent. This has an advantage over existing neural fuzzy systems because it can model more than just an `or` of `ands`, and it has an advantage over other fuzzy neuron approaches because it can be trained by gradient descent instead of set by hand.

The fuzzy logic operators listed in Section 5.2 are elegant because they are simple and continuous. However, the symmetry in these equations is difficult to see because our values are not centered about the origin. If we linearly remap these operations by defining `false` to be `-1` instead of `0`, they become:

$$
\begin{aligned}
\texttt{identity}(x) &= x \\
\texttt{not}(x) &= -x \\
\texttt{or}(x,y) &= -\tfrac{(x-1)(y-1)}{2} + 1 \\
\texttt{xor}(x,y) &= -x \cdot y \\
\texttt{and}(x,y) &= \tfrac{(x+1)(y+1)}{2} - 1 \\
\texttt{nor}(x,y) &= \tfrac{(x-1)(y-1)}{2} - 1 \\
\texttt{nxor}(x,y) &= x \cdot y \\
\texttt{nand}(x,y) &= -\tfrac{(x+1)(y+1)}{2} + 1
\end{aligned}
$$

Then, if we rewrite them in a consistent form, we obtain:

$$
\begin{aligned}
\texttt{identity}(x) &= +\left(\tfrac{(x+0)(1+0)}{1} - 0\right) \\
\texttt{not}(x) &= +\left(\tfrac{(x+0)(-1+0)}{1} - 0\right) \\
\texttt{or}(x,y) &= -\left(\tfrac{(x-1)(y-1)}{2} - 1\right) \\
\texttt{xor}(x,y) &= -\left(\tfrac{(x+0)(y+0)}{1} - 0\right) \\
\texttt{and}(x,y) &= +\left(\tfrac{(x+1)(y+1)}{2} - 1\right) \\
\texttt{nor}(x,y) &= +\left(\tfrac{(x-1)(y-1)}{2} - 1\right) \\
\texttt{nxor}(x,y) &= +\left(\tfrac{(x+0)(y+0)}{1} - 0\right) \\
\texttt{nand}(x,y) &= -\left(\tfrac{(x+1)(y+1)}{2} - 1\right)
\end{aligned}
$$

In this form, it is much more apparent that there is symmetry that can be leveraged to unify these operations into a single more general operation. There are many possible functions that perfectly express all of these fuzzy logic operations. Three representative solutions are given in Equations 5.1, 5.2, and 5.3.

$$x@y = \frac{(x+\alpha)(y+\alpha)}{\alpha^2+1} - \alpha^2 \tag{5.1}$$

$$x@y = \frac{(x+\alpha)(y+\alpha)}{|\alpha|+1} - |\alpha| \tag{5.2}$$

$$\begin{aligned} & x@y = \tfrac{t}{|t|}\sqrt{|t|} - |\alpha| \\ & \text{where } t = (x+\alpha)(y+\alpha) \end{aligned} \tag{5.3}$$

We can plug any of these functions into our table of logical operations to obtain:

$$\begin{aligned}
&\texttt{identity}(x) && = x && = \texttt{true}\textcircled{0}x \\
&\texttt{not}(x) && = -x && = \texttt{false}\textcircled{0}x \\
&\texttt{or}(x,y) && = \texttt{not}(x\textcircled{-1}y) && = \texttt{false}\textcircled{0}(x\textcircled{-1}y) \\
&\texttt{xor}(x,y) && = \texttt{not}(x\textcircled{0}y) && = \texttt{false}\textcircled{0}(x\textcircled{0}y) \\
&\texttt{and}(x,y) && = x\textcircled{1}y && \\
&\texttt{nor}(x,y) && = x\textcircled{-1}y && \\
&\texttt{nxor}(x,y) && = x\textcircled{0}y && \\
&\texttt{nand}(x,y) && = \texttt{not}(x\textcircled{1}y) && = \texttt{false}\textcircled{0}(x\textcircled{1}y)
\end{aligned}$$

Figure 5.3 shows a plot of Equation 5.2 with 5 values for α.

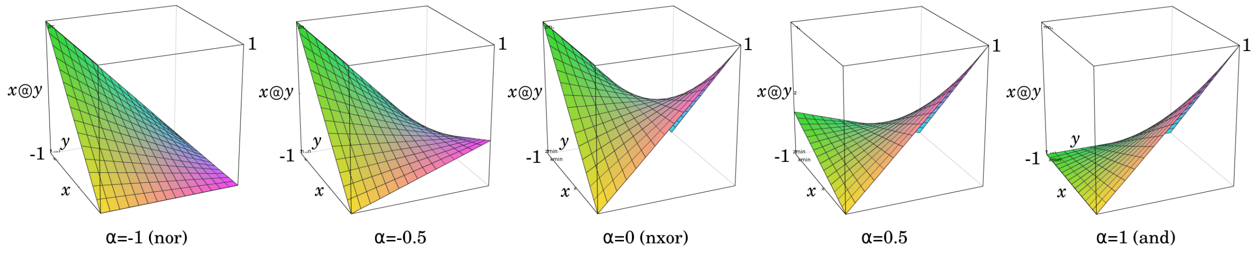

Figure 5.1: Equation 5.2 continuously interpolates among three fuzzy logic operations: `nor`, `nxor`, and `and`. By allowing biases (`true` and `false`) and weights (in particular, a weight of -1), this equation can also compute `identity`, `not`, `or`, `xor`, and `nand`.

5.4 Learning Simple Logic Operations

Since Equations 5.1, 5.2, and 5.3 are continuous and differentiable, gradient-based optimization techniques could potentially be used with them to find the values for α that approximate the logic represented in a set of training examples.

An important and consideration in using gradient descent with fuzzy logic that does not typically occur in more traditional applications for gradient descent is that each training example only provides information about a subset of the parameter space. For example, suppose α is initialized to a random value between -1 and 1, and suppose the training pattern $1@1 = 1$ is presented for optimizing the value of α by gradient descent. This training pattern suggests that α should not be less than 0, because (1 `nor` 1) $\neq 1$. However, this training pattern does not suggest anything about what specific value α should take ≥ 0, because (1 `nxor` 1) $= 1$ and (1 `and` 1) $= 1$ are both equally true.

Figure 5.4 shows a comparison of Equations 5.1, 5.2, and 5.3 for the case of computing $1@1$. If α has a value less than 0, then gradient-based optimization methods will adjust α by moving it closer to 0 no matter which of these three equations is used. As long as the curve in this region is monotonic, the precise shape is not important because these values are not defined in boolean logic. However, if Equation 5.1 is used, and α has a value greater than 0, then gradient-based optimization methods will move α closer to 0 or 1, whichever is closer to the current value of α. This is incorrect behavior because this training pattern does not provide any information about whether the logical operation should be more like `nxor` or more like `and`. Since both of these operations are consistent with the training pattern, it would be arbitrary to bias the model in favor of one over the other. Arbitrary parameter adjustments are likely to fight against subsequent training pattern presentations that may convey valid information for that region of the parameter space, resulting in the model getting stuck in a local optimum. Equations 5.2 and 5.3 correctly adjust α in all regions of the parameter space with this training pattern.

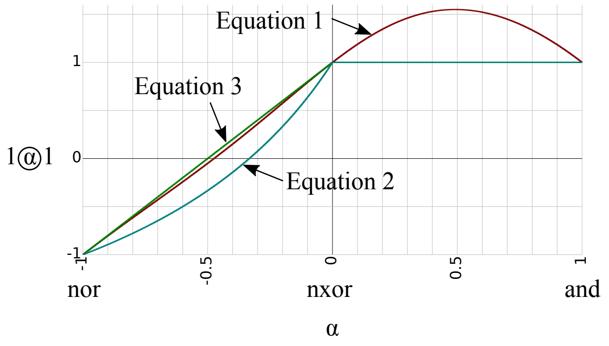

Figure 5.2: A comparison of Equations 5.1, 5.2, and 5.3 when computing `true@true`. Equation 5.1 is smoother, but Equation 5.2 is better suited for use with gradient-based optimization.

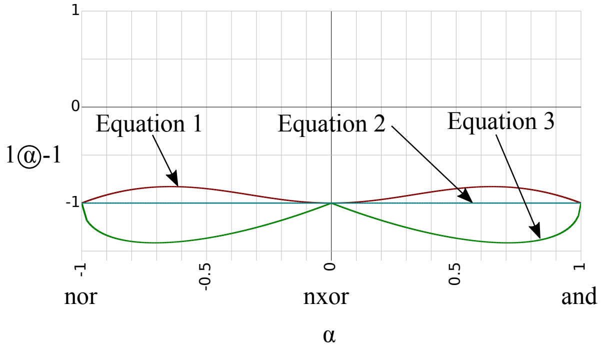

Figure 5.3: A comparison of Equations 5.1, 5.2, and 5.3 when computing `true@false`. Only Equation 5.2 is well-suited for use with gradient-based optimization.

As another representative case, consider the training pattern $1@{-1} = 1$. Perhaps counterintuitively, this pattern provides no information about any region of the parameter space, because $1\texttt{nor}{-1}$, $1\texttt{nxor}{-1}$, and $1\texttt{and}{-1}$ all evaluate to -1. Correct behavior, therefore, should not adjust the value of α, regardless of its current value. Figure 5.4 shows that only Equation 5.2 exhibits the correct behavior for this case. (This does not imply that such patterns should be discarded because in a network containing many fuzzy logic units, different input values would reach each of the units depending on the current α values.)

The remaining two cases are both mirror images of these cases, so the same analysis applies. It follows that Equation 5.2 can be expected to yield correct behavior with gradient-based optimization in all cases when training with pure boolean logic. Consistent with this intuition, we found experimentally that Equation 5.2 was always able to learning simple boolean logical expressions, while the other equations sometimes became stuck in local optima. Therefore, we used Equation 5.2 to implement $x@y$ with the remainder of our experiments.

At this point, we must note a potential problem with our approach: Equation 5.2 is not a t-norm. A t-norm must be commutative, monotonic, and associative, and the value 1 must be the identity element [61]. Although our equation is commmutative and monotonic, it is not associative and the value 1 is not neutral for all α. Our work is, therefore, not t-norm fuzzy logic but belongs instead to a broader class of fuzzy logic. For the purpose of this chapter, we use a relaxed definition of fuzzy logic as being any logic with continuous values between `true` and `false`.

5.5 Learning Complex Logic Expressions

We refer to a layer of network units that implement Equation 5.2 as a *Fuzzy* layer. Thus, Equation 5.2 is used as a sort of adaptive transfer function. The only parameters to train in a fuzzy layer are one α value per each unit. Equations 5.4, 5.5, and 5.6 give the partial derivatives of Equation 5.2, which are necessary to train such layers with gradient-based optimization methods.

$$\frac{\partial @}{\partial x} = \frac{y + \alpha}{|\alpha| + 1} \tag{5.4}$$

$$\frac{\partial @}{\partial y} = \frac{x + \alpha}{|\alpha| + 1} \tag{5.5}$$

$$\frac{\partial @}{\partial \alpha} = \frac{|a|(x + y) - a(xy + 1)}{|a|(|a| + 1)^2} \tag{5.6}$$

Most gradient-based optimization methods can be implemented in three steps: (1) A forward propagation step that computes predictions with current values, (2) a backpropagation step that computes "blame" terms for each layer in the model, and (3) an update step that refines the parameters of the model. Equation 5.2 is used in the forward propagation step. Equations 5.4 and 5.5 are used in the backpropagation step to assign blame to preceding layers. Equation 5.6 is used in the update step to refine the α values.

Because Equation 5.6 is not continuous at $\alpha = 0$, special care must be taken in the implementation of Fuzzy layers to ensure that they do not become stuck at this point. We addressed this problem in our implementation by adding the statement "**if** $\alpha < \epsilon$ **then** $\alpha \leftarrow -\alpha$" to our update step. This small addition enables α to cross over the value 0 in cases where it would otherwise become stuck. As long as ϵ is a small value, this will have negligible impact on training precision. We used the value $\epsilon = 0.001$.

Another challenge that arises in learning fuzzy logic is that Equation 5.2 only accepts two input values, x and y. One possible solution is to try to generalize the equations in a manner that can support vectors of arbitrary dimensionality. Equation 5.3 can be generalized in this manner, as given in Equation 5.7.

$$\begin{aligned} &f(\overrightarrow{x}, \alpha) = \tfrac{t}{|t|}|t|^{1/n} - |\alpha| \\ &\text{where } t = \textstyle\prod_i^n (x_i + \alpha) \end{aligned} \tag{5.7}$$

(In Equation 5.7, f is the fuzzy operator, and n refers to the number of elements in $\overrightarrow{x}$.) In higher dimensions, `or` becomes `any`, `and` becomes `all`, and `xor` becomes `parity`. Unfortunately, Equation 5.3 resists gradient-based optimization, so its generalized version is unlikely to do any better, and Equation 5.2 cannot be generalized in this manner. Further, in applications with many variables, it is often unlikely that all variables will simultaneously take the same value, which renders the `any` and `all` operations to have very limited utility. Therefore, a good solution should provide a mechanism to select which values feed into each logical operation. This is also consistent with most real-world uses of boolean logical expressions, and logical expressions involving only two variables at a time are more likely to be easily comprehensible to humans than logic involving many values. Our solution to this challenge introduces two additional layer types, which we call *AllPairings*, and *FeatureSelector*.

An AllPairings layer accepts n inputs, and outputs all $n(n-1)/2$ possible unordered pairings of its input values. In order to facilitate the `identity` and `not` operations, we additionally pair each input value with the bias values `true` and `false`, which increases the total number of unordered output pairs to $n(n-1)/2+2n$. This layer type contains no parameters to train, so it is straightforward to use in a deep network. During the backpropagation step, the blame term assigned to each input unit is simply the sum of the blame terms for all affect output units.

A FeatureSelector layer is identical to a traditional fully-connected linear layer, except with four minor modifications: (1) No bias weights are used, (2) The weights are initialized with uniform values instead of random values, (3) The weights that feed into each unit are constrained to have values between -1 and 1, and (4) L^1 regularization is applied to the weights in this layer to gently promote sparse connections in this layer. As it is closely related to a fully-connected layer, it produces a weighted sum of its inputs. This allows the network to learn an interpolation between logical expressions learned in the Fuzzy layers. The outputs of a FeatureSelector layer can be re-mapped between -1 and 1 (i.e. through a membership function) before they are used as input to other layers.

Our topology, given the definitions of these two layer types, is as follows. Given n continuous input values, we first normalize them between -1 and 1; this can be thought of as a single linear membership function (where 1 is "high" and -1 is "low"). Next, we feed the normalized values into an AllPairings layer. We then feed all combinations of value pairings into a Fuzzy layer, which learns an optimal logical operation for each pair of values. The output of the Fuzzy layer is fed into a FeatureSelector layer to manage dimensionality and to produce the desired number of output values. If a deeper topology is desired, we can feed the output of the FeatureSelector layer into another normalizing layer (i.e. a single membership

Table 5.1: Average errors of a deep neural network (DNN), an ensemble of ANFIS fuzzy classifiers (EFC), our model, and the expressions obtained from "snapping" the weights in our network (Snapped) on the validation data for five classification problems. Best results are **bolded**.

Dataset	DNN	EFC	Our Model	Snapped
Breast Cancer	3.26%	6.42%	**2.77%**	2.84%
Diabetes	29.68%	24.51%	**22.79%**	35.06%
Vehicle	**18.01%**	50.58%	28.71%	67.84%
Waveform	**14.95%**	-	15.27%	68.43%
Yeast	**46.12%**	67.37%	49.77%	82.94%

function for each output) and repeat the sequence of AllPairings, Fuzzy, and FeatureSelector layers to an arbitrary depth. If no further depth is needed, we can feed the output of the final FeatureSelector layer into any kind of output layer; we use a `max` layer for classification.

In our validation, we fix the depth of logic layer to two, resulting in the following final 10-layer deep topology: (1) the input layer (identity), (2) a normalization layer (a single linear membership function), (3) an AllPairings layer, (4) a Fuzzy layer, (5) a FeatureSelector layer, (6) a `tanh` layer (a single nonlinear membership function with a new input space), (7) another AllPairings layer, (8) another Fuzzy layer, (9) another FeatureSelector layer, and (10) a `max` layer for classification.

5.6 Validation

Our goal in this work is not to surpass the accuracy of existing results or to claim any novelty in learning fuzzy logic rules, but to demonstrate an alternative approach to fuzzy learning using a novel adaptive transfer function. We apply our model to five classification problems taken from the UCI Machine Learning Repository [5], comparing it with a regular deep neural network (DNN) with `tanh` activation functions as well as with ensemble of fuzzy classifiers (EFC) proposed by Canul-Reich et. al in 2007 [13]. Our results demonstrate that our model meets our goal, adaptively learning complex logic expressions by gradient descent while yielding accuracies comparable to existing methods and learning fuzzy logic rules.

The five classification problems we used were breast cancer, diabetes, vehicle, waveform, and yeast. Errors yielded by four models are compared in Table 5.1. Across all five problems, we fixed our models parameters to demonstrate robustness. In particular, we set the learning rate to 0.01 and the regularization term to 0.0001. We use a DNN topology with a depth and number of weights similar to our model, but with each layer being fully connected and all activation functions set to `tanh`.

We also include the results reported by Canul-Reich et. al [13] rather than re-evaluating their method ourselves (which is why there is no result for their model on the waveform problem). Their model is an ANFIS-based ensemble of fuzzy classifiers (labeled EFC in Table 5.1). One of the important observations made by Canul-Reich et. al was that their model performed poorly on datasets with more than six input features (vehicle and yeast).

All of the datasets we use have continuous inputs and discrete class outputs. Breast cancer has 9 inputs and 2-class outputs. Our model outperforms both DNN and EFC in terms of accuracy. Diabetes (Pima) has 8 inputs and 2-class outputs. Again, our model achieves higher accuracy than the compared models.

Vehicle has 18 inputs and 4-class outputs. This is an interesting problem because it has a high number of inputs and more than a binary output. The DNN has significantly lower error than our model, likely due to the difficulties fuzzy systems typically have with larger input spaces. However, our model makes more accurate predictions than EFC, demonstrating that although our model is susceptible to problems with large input spaces, it is more robust than some other fuzzy-based approaches.

Waveform has 40 inputs and 3-class outputs. This is a very large input space, so it is an interesting test for a fuzzy-based system like ours. Our model performed well on this problem, yielding test errors only marginally higher than the DNN. (Canul-Reich et. al did not report results on this problem for EFC).

Yeast has 8 inputs and 10-class outputs. Out of the five datasets we used, this had the highest number of output classes, so the results of this test demonstrate how our model

Table 5.2: Logical expressions learned by our model for three of the datasets. These expressions were formed by “snapping” all parameters of the network to the nearest whole value. Expressions for the other two datasets were omitted for the sake of brevity.

Dataset	Expression
Breast Cancer	$c_0 = (0 \mid 3) + (1 \mid 5)$ $c_1 = ((1 \mid 3) + (3 \mid 7)) \mid ((2 \mid 8) + (3 \mid 4) + (5 \mid 6))$
Diabetes	$c_0 = (\neg(2 \mid 4) + 7 + \neg(2 \mid 6)) \mid (\neg(2 \,\&\, 6) + \neg 0)$ $c_1 = (\neg(2 + \neg(1 \,\&\, 5) + 1) \,\&\, (2 \mid 6))$
Vehicle	$c_0 = \neg((0 \,\&\, 4) \mid ((6 \,\&\, 10) + (4 \,\&\, 16) + (7 \,\&\, 13)))$ $\quad + (\neg(2 \mid 16) \,\&\, ((4 \,\&\, 9) + (5 \,\&\, 6)))$ $c_1 = (4 \,\&\, 9) + (5 \,\&\, 6) + \neg(((0 \,\&\, 4)) \oplus (8 \,\&\, 16))$ $c_2 = \neg(((3 \,\&\, 5) + (0 \,\&\, 4) + (3 \,\&\, 13)) \,\&\, (0 \,\&\, 4))$ $c_3 = \neg((5 \,\&\, 15) \,\&\, (0 \,\&\, 4))$

handles many classes. Our model performed nearly as well as the DNN and significantly better than the EFC, demonstrating that it is able to model data with many partitions.

Our model’s weights can be interpreted as complex logical expressions that describe the relationships learned between various inputs. We form these expressions by “snapping” the α parameters to the nearest whole value (i.e. $\alpha = -0.2$ snaps to 0 to become `nxor`) and “snapping” the weights on the FeatureSelector layer to the nearest whole value (effectively negating expressions with weights near -1 and dropping expressions with weights near 0). We use n to denote that input n is “high”, $\neg$ for `not`, & for `and`, | for `or`, and $\oplus$ for `xor`. c_i is the output for class i, where the i with maximum c_i is the predicted class. Addition has no equivalent logical operation, so we leave it as a sum indicating the interpolation between operands. (In the network, the sum is fed through a `tanh` function to map the result back into our logical space; this step is omitted in the snapped expressions). The resulting expressions for three of the problems (breast cancer, diabetes, and vehicle) are shown in Table 5.2; expressions for waveform and yeast are omitted from this chapter for brevity. Accuracies of the formed expressions applied to the validation data are reported in the “Snapped” column of Table 5.1.

5.7 Conclusion

We presented a deep learning architecture for learning fuzzy logic expressions. The primary component of this architecture is an innovative, parameterized, differentiable activation function that can learn a number of logical operations by gradient descent. This activation function is unique to our approach and unifies fuzzy logic with deep learning in a new way that leverages the advantages of both domains. We provided both a theoretical basis for our activation function and testing results on five classification problems from the UCI Machine Learning Repository. Not only was our model a reasonably strong classifier for these problems, but also the parameters of our model were interpretable as logical expressions that summarized what it had learned. Our primary contribution is neither a better classifier nor the first model whose weights can be interpreted as fuzzy logic rules, however, but the adaptive transfer function that learns logical operations by gradient descent. We believe that this is a promising new approach to combining fuzzy logic with deep learning that potentially opens the way for future work in both domains.

Chapter 6

Neural Decomposition of Time-Series Data

Abstract: We present a neural network technique for the analysis and extrapolation of time-series data called Neural Decomposition (ND). Units with a sinusoidal activation function are used to perform a Fourier-like decomposition of training samples into a sum of sinusoids, augmented by units with nonperiodic activation functions to capture linear trends and other nonperiodic components. We show how careful weight initialization can be combined with regularization to form a simple model that generalizes well. Our method generalizes effectively on the Mackey-Glass series, a dataset of unemployment rates as reported by the U.S. Department of Labor Statistics, a time-series of monthly international airline passengers, the monthly ozone concentration in downtown Los Angeles, and an unevenly sampled time-series of oxygen isotope measurements from a cave in north India. We find that ND outperforms popular time-series forecasting techniques including LSTM, echo state networks, ARIMA, SARIMA, SVR with a radial basis function, and Gashler and Ashmore's model.

6.1 Introduction

The analysis and forecasting of time-series is a challenging problem that continues to be an active area of research. Predictive techniques have been presented for an array of problems, including weather [45], traffic flow [112], seizures [37], sales [23], and others [155, 80, 19, 156]. Because research in this area can be so widely applied, there is great interest in discovering more accurate methods for time-series forecasting.

One approach for analyzing time-series data is to interpret it as a signal and apply the Fourier transform to decompose the data into a sum of sinusoids [8]. Unfortunately, despite the well-established utility of the Fourier transform, it cannot be applied directly to time-series forecasting. The Fourier transform uses a predetermined set of sinusoid frequencies rather than learning the frequencies that are actually expressed in the training data. Although the signal produced by the Fourier transform perfectly reproduces the training samples, it also predicts that the same pattern of samples will repeat indefinitely. As a result, the Fourier transform is effective at interpolation but is unable to extrapolate future values. Another limitation of the Fourier transform is that it only uses periodic components, and thus cannot accurately model the nonperiodic aspects of a signal, such as a linear trend or nonlinear abnormality.

Another approach is regression and extrapolation using a model such as a neural network. Regular feedforward neural networks with standard sigmoidal activation functions do not tend to perform well at this task because they cannot account for periodic components in the training data. Fourier neural networks have been proposed, in which feedforward neural networks are given sinusoidal activation functions and are initialized to compute the Fourier transform. Unfortunately, these models have proven to be difficult to train [45].

Recurrent neural networks, as opposed to feedforward neural networks, have been successfully applied to time-series prediction [46, 49]. However, these kinds of networks make up a different class of forecasting techniques. Recurrent neural networks also have difficulty handling unevenly sampled time-series. Further discussion about recurrent neural networks and other classes of forecasting techniques is provided in Section 6.2.

We claim that effective generalization can be achieved by regression and extrapolation using a model with two essential properties: (1) it must combine both periodic and nonperiodic components, and (2) it must be able to tune its components as well as the weights used to combine them. We present a neural network technique called Neural Decomposition (ND) that demonstrates this claim. Like the Fourier transform, it decomposes a signal into a sum of constituent parts. Unlike the Fourier transform, however, ND is able to reconstruct a signal that is useful for extrapolating beyond the training samples. ND trains the components into which it decomposes the signal represented by training samples. This enables it to find a simpler set of constituent signals. In contrast to the fast Fourier transform, ND does not require the number of samples to be a power of two, nor does it require that samples be measured at regular intervals. Additionally, ND facilitates the inclusion of nonperiodic components, such as linear or sigmoidal components, to account for trends and nonlinear irregularities in a signal.

In Section 6.5, we demonstrate that the simple innovations of ND work together to produce significantly improved generalizing accuracy with several problems. We tested with the chaotic Mackey-Glass series, a dataset of unemployment rates as reported by the U.S. Department of Labor Statistics, a time-series of monthly international airline passengers, the monthly ozone concentration in downtown Los Angeles, and an unevenly sampled time-series of oxygen isotope measurements from a cave in north India. We compared against long short-term memory networks (LSTM), echo state networks, autoregressive integrated moving average (ARIMA) models, seasonal ARIMA (SARIMA) models, support vector regression with a radial basis function (SVR), and a model recently proposed by Gashler and Ashmore [45]. In all but one case, ND made better predictions than each of the other prediction techniques evaluated; in the excepted case, LSTM and echo state networks performed slightly better than ND.

This chapter is outlined as follows. Section 6.2 provides a background and reviews related works. Section 6.3 gives an intuitive-level overview of ND. Section 6.4 provides finer implementation-level details. Section 6.5 shows results that validate our work. Finally, Section 6.6 discusses the contributions of this chapter and future work.

6.2 Related Work

6.2.1 Models for Time-Series Prediction

Many works have diligently surveyed the existing literature regarding techniques for forecasting time-series data [31, 90, 176, 15, 42, 175, 30]. Some popular statistical models include Gaussian process [10] and hidden Markov models [116].

Autoregressive integrated moving average (ARIMA) models [177, 164] are among the

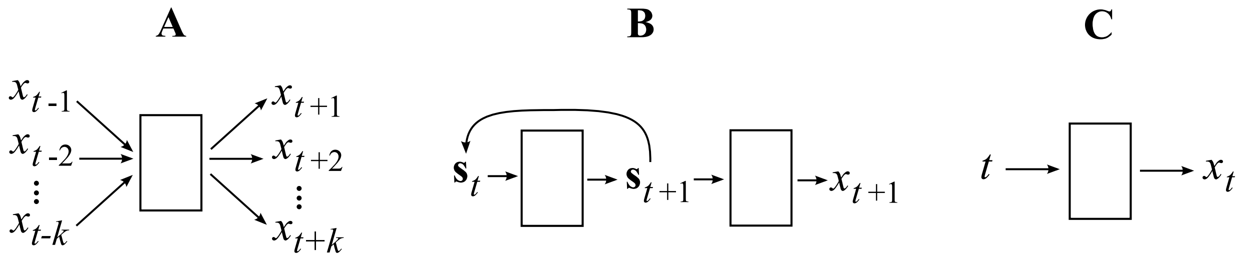

Figure 6.1: Three broad classes of models for time-series forecasting: (A) prediction using a sliding window, (B) recurrent models, and (C) regression-based extrapolation.

most popular approaches. The notation for this model is ARIMA(p, d, q), where p is the number of terms in the autoregressive model, d is the number of differences required to take to make the time-series stationary, and q is the number of terms in the moving average model. In other words, ARIMA models compute the dth difference of $x(t)$ as a function of $x_{t-1}, x_{t-2}, ..., x_{t-p}$ and the previous q error terms.

Out of all the ARIMA variations that have been proposed, seasonal ARIMA (SARIMA) [9] is considered to be the state of the art "classical" time-series approach [112]. Notation for SARIMA is ARIMA$(p, d, q)(P, D, Q)[S]$, where p, d, q are identical to the normal ARIMA model, P, D, Q are analogous seasonal values, and S is the seasonal parameter. For example, an ARIMA(1,0,1)(0,1,1)[12] uses an autoregressive model with one term, a moving average model with one term, one seasonal difference (that is, $x'_t = x_t - x_{t-12}$), and a seasonal moving average with one term. This seasonal variation of ARIMA exploits seasonality in data by correlating x_t not only with recent observations like x_{t-1}, but also with seasonally recent observations like x_{t-S}. For example, when the data is a monthly time-series, $S = 12$ correlates observations made in the same month of different years, and when the data is a daily time-series, $S = 7$ correlates observations made on the same day of different weeks.

In the field of machine learning, three high-level classes of techniques (illustrated in Figure 6.1) are commonly used to forecast time-series data [45]. Perhaps the most common approach, (A), is to train a model to directly forecast future samples based on a sliding window of recently collected samples [42]. This approach is popular because it is simple to implement and can work with arbitrary supervised learning techniques.

A more sophisticated approach, (B), is to train a recurrent neural network [89, 126]. Several recurrent models, such as LSTM networks [46, 49], have reported very good results for forecasting time-series. In an LSTM network, each neuron in the hidden layer has a memory cell protected by a set of gates that control the flow of formation through time[49]. Echo state networks (ESNs) have also performed particularly well at this task [106, 82, 140]. An ESN is a randomly connected, recurrent reservoir network with three primary meta-parameters: input scaling, spectral radius, and leaking rate [114]. Although they are powerful, these recurrent models are only able to handle time-series that are sampled at a fixed interval, and thus cannot be directly applied to unevenly sampled time-series.

Our model falls into the third category of machine learning techniques, (C): regression-based extrapolation. Models of this type fit a curve to the training data, then use the trained curve to anticipate future samples. One advantage of this approach over recurrent neural networks is that it can make continuous predictions, instead of predicting only at regular intervals, and can therefore be directly applied to irregularly spaced time-series. A popular method in this category is support vector regression (SVR) [150, 32]. Many models in this category decompose a signal into constituent parts, providing a useful mechanism for analyzing the signal. Our model is more closely related to a subclass of methods in this category, called Fourier neural networks (see Section 6.2.3), due to its use of sinusoidal activation functions. Models in the first two categories, (A) and (B), have already been well-studied, whereas extrapolation with sinusoidal neural networks remains a relatively unexplored area.

6.2.2 Harmonic Analysis

The harmonic analysis of a signal transforms a set of samples from the time domain to the frequency domain. This is useful in time-series prediction because the resulting frequencies can be used to reconstruct the original signal (interpolation) and to forecast values beyond the sampled time window (extrapolation). Harmonic analysis, also known as spectral analysis or spectral density estimation, has been well-studied for decades [149, 131, 39, 128].

Perhaps the most popular method of harmonic analysis is the distrete Fourier transform (DFT). The DFT maps a series of N complex numbers in the time domain to the frequency domain. The inverse DFT (iDFT) can be applied these new values to map them back to the time domain. More interestingly, the iDFT can be used as a continuous representation of the originally discrete input. The transforms are generally written as a sum of N complex exponentials, which can be rewritten in terms of sines and cosines by Euler's formula.

The DFT and the iDFT are effectively the same transform with two key differences. First, in terms of sinusoids, the DFT uses negative multiples of $2\pi/N$ as frequencies and the iDFT uses positive multiples of $2\pi/N$ as frequencies. Second, the iDFT contains the normalization term $1/N$ applied to each sum.

In general, the iDFT requires all N complex values from the frequency domain to reconstruct the input series. For real-valued input, however, only the first $N/2+1$ complex values are necessary ($N/2$ frequencies and one bias). The remaining complex numbers are the conjugates of the first half of the values, so they only contain redundant information. Furthermore, in the real-valued case, the imaginary component of the iDFT output can be discarded to simplify the equation, as we do in Equation 6.1. This particular form of the iDFT (reconstructing a series of real samples) can therefore be written as a real sum of sines and cosines.

The iDFT is as follows. Let R_k and I_k represent the real and imaginary components respectively of the kth complex number returned by the DFT. Let $2\pi k/N$ be the frequency of the kth term. The first frequency yields the bias, because $cos(0) = 1$ and $sin(0) = 0$. The second frequency is a single wave, the third frequency is two waves, the fourth frequency is three waves, and so on. The cosine with the kth frequency is scaled by R_k, and the sine with the kth frequency is scaled by I_k. Thus, the iDFT is sufficiently described as a sum of $N/2+1$ terms, with a $sin(t)$ and a $cos(t)$ in each term and a complex number from the

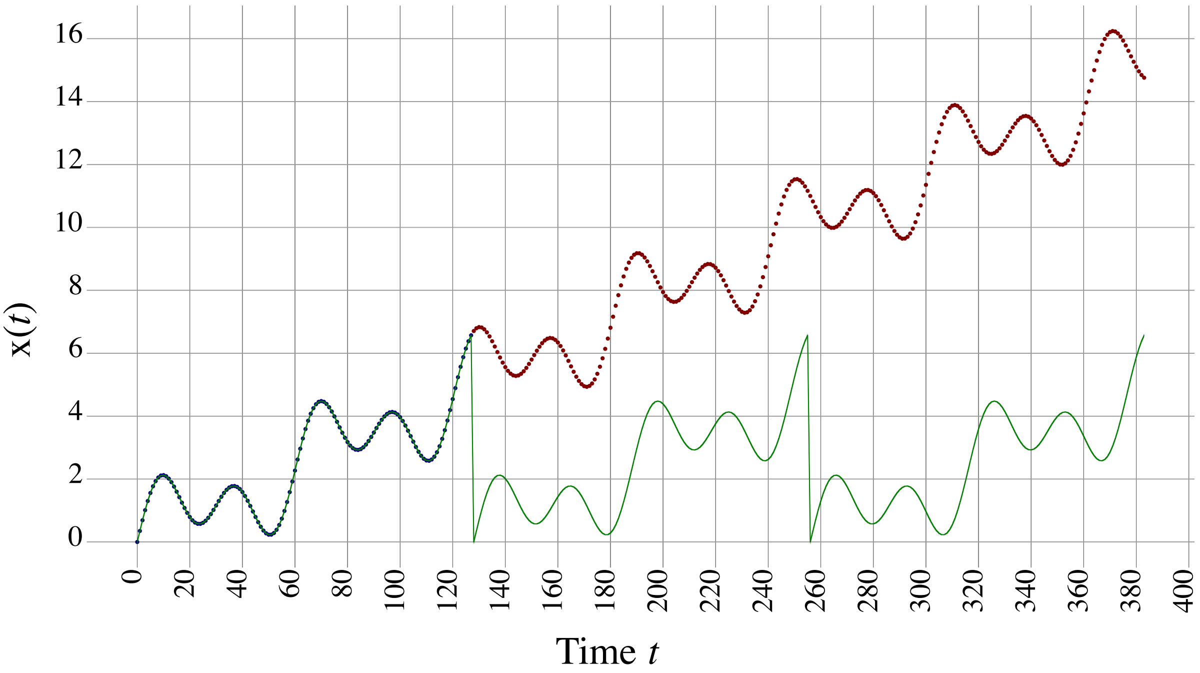

Figure 6.2: The predictive model generated by the iDFT for a toy problem with both periodic and nonperiodic components. Blue dots represent training samples, red dots represent testing samples, and the green line represents the iDFT. Two significant problems limit its ability to generalize: (1) The model repeats, ignoring the linear trend, and (2) The extrapolated predictions misalign with the phase of the continuing nonlinear trend.

DFT corresponding to each term:

$$x(t) = \sum_{k=0}^{N/2} R_k \cdot cos(\frac{2\pi k}{N} t) - I_k \cdot sin(\frac{2\pi k}{N} t) \tag{6.1}$$

Equation 6.1 is useful as a continuous representation of the real-valued discrete input. Because it perfectly passes through the input samples, one might naively expect this function to be a good basis for generalization. In order to choose appropriate frequencies, however, the iDFT assumes that the underlying function always has a period equal to the size of the samples that represent it, that is, $x(t + N) = x(t)$ for all t. Typically, in cases where generalization is desirable, the period of the underlying function is not known. The iDFT cannot effectively model the nonperiodic components of a signal, nor can it form a simple model for series that are not periodic at N, even if the series is perfectly periodic.

Figure 6.2 illustrates the problems encountered when using the iDFT for time-series forecasting. Although the model generated by the iDFT perfectly fits the training samples, it only has periodic components and so is only able to predict that these samples will repeat to infinity, without taking nonperiodicity into account. Our approach mimics the iDFT for modeling periodic data, but is also able to account for nonperiodic components in a signal (Figure 6.4).

Because of these limitations of the DFT, other approaches to the harmonic analysis of time-series have been proposed. Some of these other approaches perform sinusoidal regression to determine frequencies that better represent the periodicity of the sampled signal [162, 168]. Our approach similarly uses regression to find better frequencies.

6.2.3 Fourier Neural Networks

Use of the Fourier transform in neural networks has already been explored in various contexts [69, 182]. The term *Fourier neural network* has been used to refer to neural networks that use a Fourier-like neuron [142], that use the Fourier transform of some data as input [123], or that use the Fourier transform of some data as weights [45]. Our work is not technically a Fourier neural network, but of these three types, our approach most closely resembles the third.

Silvescu provided a model for a Fourier-like activation function for neurons in neural networks [142]. His model utilizes every unit to form DFT-like output for its inputs. He notes that by using gradient descent to train sinusoid frequencies, the network is able to learn “exact frequency information” as opposed to the “statistical information” provided by the DFT. Our approach also trains the frequencies of neurons with a sinusoidal activation function.

Gashler and Ashmore presented a technique that used the fast Fourier transform (FFT) to approximate the DFT, then used the obtained values to initialize the sinusoid weights of a neural network that mixed sinusoidal, linear, and softplus activation functions [45]. Because this initialization used sinusoid units to model nonperiodic components of the data, their model was designed to heavily regularize sinusoid weights so that as the network was trained, it gave preference to weights associated with nonperiodic units and shifted the weights from the sinusoid units to the linear and softplus units. Use of the FFT required their input size to be a power of two, and their trained models were slightly out of phase with their validation data. However, they were able to generalize well for certain problems. Our approach is similar, except that we do not use the Fourier transform to initialize any weights (further discussion on why we do not use the Fourier transform can be found in Section 6.3.3).

6.3 High Level Approach

In this section, we describe Neural Decomposition (ND), a neural network technique for the analysis and extrapolation of time-series data. This section focuses on an intuitive-level overview of our method; implementation details can be found in Section 6.4.

6.3.1 Algorithm Description

We use an iDFT-like model with two simple but important innovations. First, we allow sinusoid frequencies to be trained. Second, we augment the sinusoids with a nonperiodic function to model nonperiodic components. The iDFT-like use of sinusoids allows our model to fit to periodic data, the ability to train the frequencies allows our model to learn the true period of a signal, and the augmentation function enables our model to forecast time-series that are made up of both periodic and nonperiodic components.

Our model is defined as follows. Let each a_k represent an amplitude, each w_k represent a frequency, and each ϕ_k represent a phase shift. Let t refer to time, and let $g(t)$ be an augmentation function that represents the nonperiodic components of the signal. Our model is defined as

$$x(t) = \sum_{k=1}^{N} \big(a_k \cdot sin(w_k t + \phi_k)\big) + g(t). \tag{6.2}$$

Note that in our model, compared to the iDFT, two indexing changes have been made: 1) the lower index of the sum has changed from $k = 0$ to $k = 1$, and 2) the upper index of the sum has changed from $N/2$ to N. The lower index has changed because ND can account for bias in the augmentation function $g(t)$, so the 0 frequency is not necessary. The upper index has changed to simplify the equation as a sum of N sines rather than a sum of $N/2$ sines and cosines.

If the phase shifts are set so that $sin(t + \phi)$ is transformed into $cos(t)$ and $-sin(t)$, the frequencies are set to the appropriate multiples of 2π, the amplitudes are set to the output values of the DFT, and $g(t)$ is set to a constant (the bias), then ND is identical to the iDFT. However, by choosing a $g(t)$ better suited to generalization and by learning the amplitudes and tuning the frequencies using backpropagation, our method is more effective at generalization than the iDFT. $g(t)$ may be as simple as a linear equation or as complex as a combination of linear and nonlinear equations. A discussion on the selection of $g(t)$ can be found in Section 6.4.

We use a feedforward artificial neural network with a single hidden layer to compute our function (see Figure 6.3). The hidden layer is composed of N units with a sinusoid activation function and an arbitrary number of units with other activation functions to calculate $g(t)$. The output layer is a single linear unit, so that the neural network outputs a linear combination of the units in the hidden layer.

We initialize the frequencies and phase shifts in the same way as the inverse DFT as described above. Rather than use the actual values provided by the DFT as sinusoid amplitudes, however, we initialize them to small random values (see Section 6.3.3 for a discussion on why). Weights in the hidden layer associated with $g(t)$ are initialized to approximate identity, and weights in the output layer associated with $g(t)$ are randomly perturbed from zero.

We train our model using stochastic gradient descent with backpropagation. This training process allows our model to learn better frequencies and phase shifts so that the sinusoid units

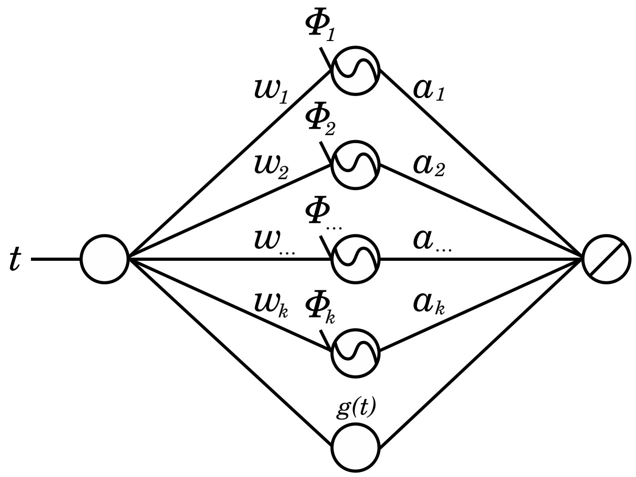

Figure 6.3: A diagram of the neural network model used by Neural Decomposition. For each of the k sinusoid units, w_i are frequencies, ϕ_i are phase shifts, and a_i are amplitudes, where $i \in \{1 \ldots k\}$. The augmentation function $g(t)$ is shown as a single unit, but it may be composed of one or more units with one or more activation functions.

more accurately represent the periodic components of the time-series. Because frequencies and phase shifts are allowed to change, our model can learn the true period of the underlying function rather than assuming the period is N. Training also tunes the weights of the augmentation function.

ND uses regularization throughout the training process to distribute weights in a manner consistent with our goal of generalization. In particular, we use L^1 regularization on the output layer of the network to promote sparsity by driving nonessential weights to zero. Thus, ND produces a simpler model by using the fewest number of units that still fit the training data well.

By pre-initializing the frequencies and phase shifts to mimic the inverse DFT and setting all other parameters to small values, we reduce time-series prediction to a simple regression problem. Artificial neural networks are particularly well-suited to this kind of problem, and using stochastic gradient descent with backpropagation to train it should yield a precise and accurate model.

The neural network model and training approach we use is similar to those used by Gashler and Ashmore in a previous work on time-series analysis [45]. Our work builds on

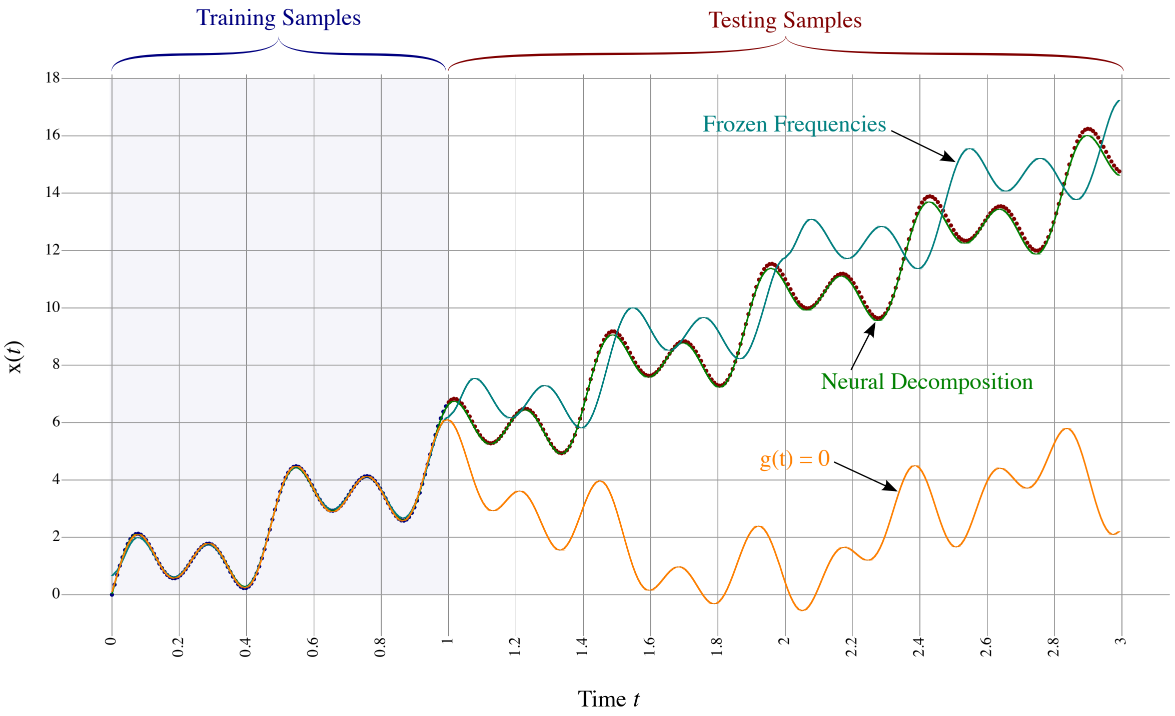

Figure 6.4: A comparison of Neural Decomposition with two algorithmic variations showing the importance of certain algorithm details. The data used here is the same data used in Figure 6.2. The full ND model, shown in green, fits very closely to the data that was withheld during training. The cyan curve shows predictions made when the basis functions, including sinusoidal frequencies, were frozen during training. Note that the predictions are out-of-phase, indicating that training these components is essential for effective generalization. The orange curve shows predictions made without including any nonperiodic components among the basis functions, that is, setting the augmentation function $g(t) = 0$. Although the predictions exhibit the correct phase, they fail to fit with the nonperiodic trend. This shows the importance of using heterogeneous basis functions.

theirs and contributes a number of improvements, both theoretically and practically. First, we do not initialize the weights of the network using the Fourier transform. This proved to be problematic in their work as it used periodic components to model linear and other nonperiodic parts of the training data. By starting with weights near zero and learning weights for both periodic and nonperiodic units simultaneously, our model does not have to unlearn extraneous weights. Second, their model required heavy regularization that favored using linear units rather than the initialized sinusoid units. Our training process makes no assumptions about which units are more important and instead allows gradient descent to determine which components are necessary to model the data. Third, their training process required a small learning rate (on the order of 10^{-7}) and their network was one layer deeper than ours. As a result, their frequencies were never tuned, their results were generally out of phase with the testing data, and their training times were very long. Because our method facilitates the training of each frequency and allows a larger learning rate (10^{-3} in our experiments), our method yields a function that is more precisely in phase with the testing data in a much shorter amount of time. Thus, our method has simplified the complexity of the model's training algorithm, minimized its training time, and improved its overall effectiveness at time-series prediction. The superiority of our method is demonstrated in Section 6.5 and visualized in Figure 6.8.

6.3.2 Toy Problem for Justification

Figure 6.4 demonstrates that flexible frequencies and an appropriate choice for $g(t)$ are essential for effective generalization. We compare three ND models using the equation $x(t) = sin(4.25\pi t) + sin(8.5\pi t) + 5t$ to generate time-series data. This is a sufficiently interesting toy problem because it is composed of periodic and nonperiodic functions and its period is not exactly N (otherwise, the frequencies would have been multiples of 2π). We generate 128 values for $0 \leq t < 1.0$ as input and 256 values for $1.0 \leq t < 3.0$ as a validation set. Powers of two are not required, but we used powers of two in order to compare our approach with using the inverse DFT (approximated by the inverse FFT).

One of the compared ND models freezes the frequencies so that the model is unable to adjust them. Although it is able to find the linear trend in the signal, it is unable to learn the true period of the data and, as a result, makes predictions that are out of phase with the actual signal. This demonstrates that the ability to adjust the constituent parts of the output signal is necessary for effective generalization.

Another of the compared ND models has flexible frequencies, but uses no augmentation function (that is, $g(t) = 0$). This model can learn the periodic components of the signal, but

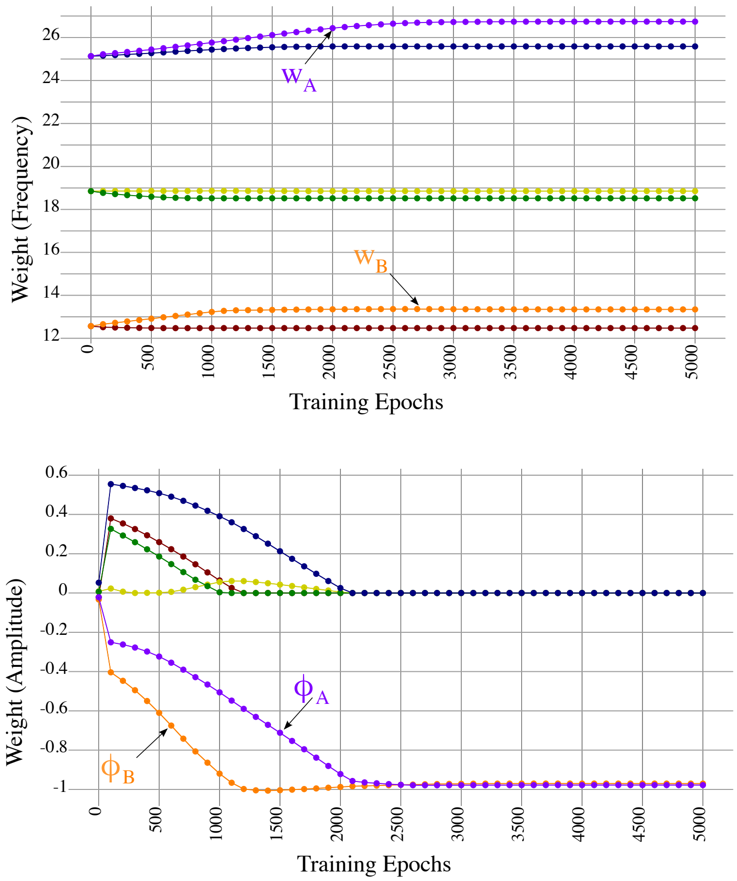

Figure 6.5: (Top) Frequencies of the basis functions of Neural Decomposition over time. (Bottom) Basis weights (amplitudes) over time on the same problem. Note that ND first tunes the frequencies (Top), then finishes adjusting the corresponding amplitudes for those sinusoids (Bottom) (w_A corresponds to ϕ_A and w_B corresponds to ϕ_B). In most cases, the amplitudes are driven to zero to form a sparse representation. After the amplitudes reach zero, the frequencies are no longer modified.

not its nonperiodic trend. It tunes the frequencies of the sinusoid units to more accurately reflect the input samples, so that it is more in phase than the second model. However, because it cannot explain the nonperiodic trend of the signal, it also uses more sinusoid units than the true underlying function requires, resulting in predictions that are not perfectly in phase. This model shows the necessity of an appropriate augmentation function for handling nonperiodicity.

The final ND model compared in Figure 6.4 is ND with flexible frequencies and augmentation function $g(t) = wt+b$. As expected, it learns both the true period and the nonperiodic

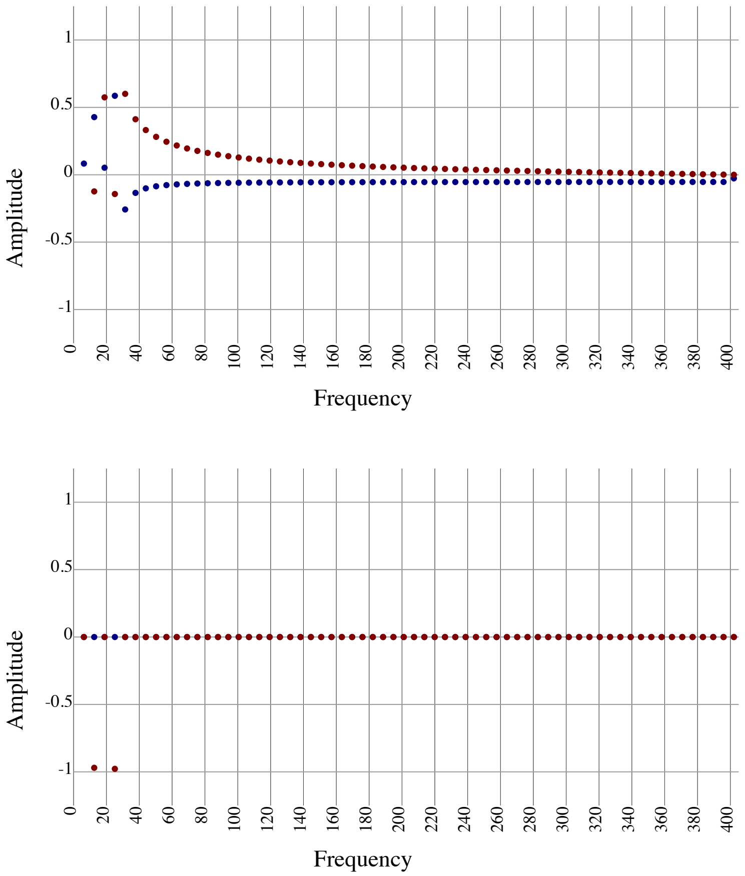

Figure 6.6: Frequency domain representations of the toy problem (amplitude vs frequency). (Top) Frequencies used by the iDFT. (Bottom) Frequencies used by ND.

trend of the signal. We therefore conclude that an appropriate augmentation function and the ability to tune components are essential in order for ND to generalize well.

6.3.3 Toy Problem Analysis

In Figure 6.5, we plot the weights over time of our $g(t) = wt + b$ model being trained on the toy problem. Weights in Figure 6.5(a) are the frequencies of a few of the sinusoids in the model, initialized based on the iDFT, but tuned over time to learn more appropriate frequencies for the input samples, and weights in Figure 6.5(b) are their corresponding amplitudes. The training process tunes frequencies w_A and w_B to more accurately reflect the period of the underlying function and adjusts the corresponding amplitudes ϕ_A and ϕ_B so that only the sinusoids associated with these amplitudes are used in the trained model

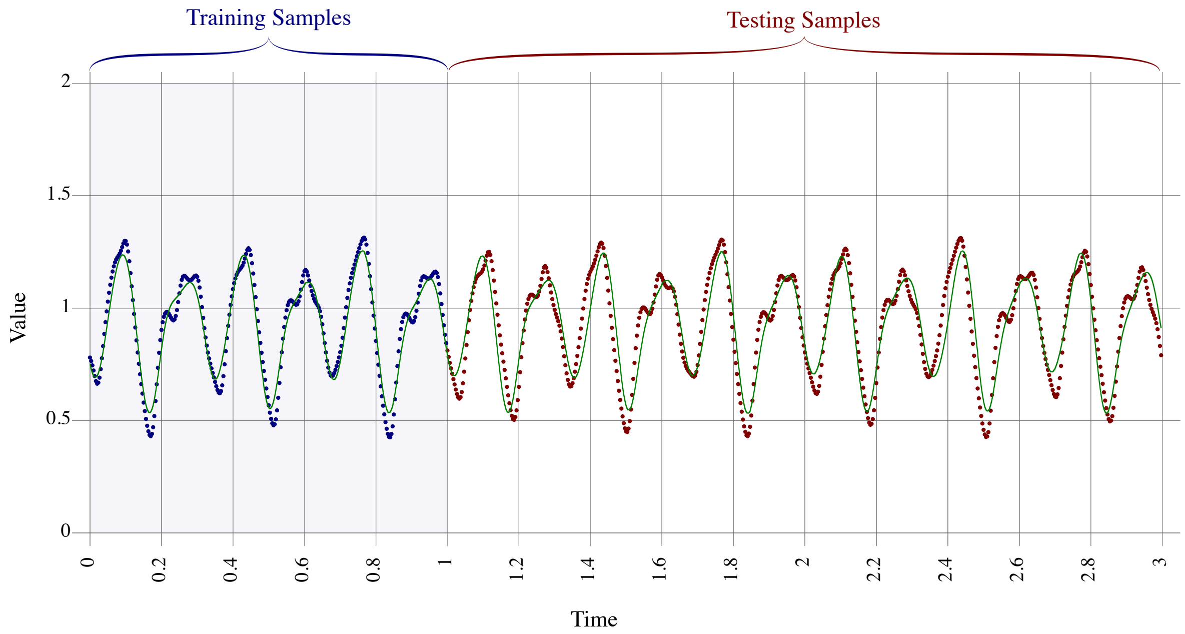

Figure 6.7: Neural Decomposition on the Mackey-Glass series. Although it does not capture all the high-frequency fluctuations in the data, our model predicts the location and height of each peak and valley in the series with a high degree of accuracy.

and all other amplitudes are driven to zero. This demonstrates that ND tunes frequencies it needs and learns amplitudes as we hypothesized. It is also worth noting that after the first 2500 training epochs, no further adjustments are made to the weights. This suggests that ND is robust against overfitting, at least in some cases, as the "extra" training epochs did not result in a worse prediction.

Gashler and Ashmore utilized the FFT to initialize the sinusoid amplitudes so that the neural network immediately resembled the iDFT [45]. Using the DFT in this way yields an unnecessarily complex model in which nearly every sinusoid unit has a nonzero amplitude, either because it uses periodic functions to model the nonperiodic signal or because it has fixed frequencies and so uses a range of frequencies to model the actual frequencies in the signal [142]. Consequently, the training process required heavy regularization of the sinusoid amplitudes in order to shift the weight to the simpler units (see Section 6.2.3). Training from this initial point often fell into local optima, as such a model was not always able to unlearn superfluous sinusoid amplitudes.

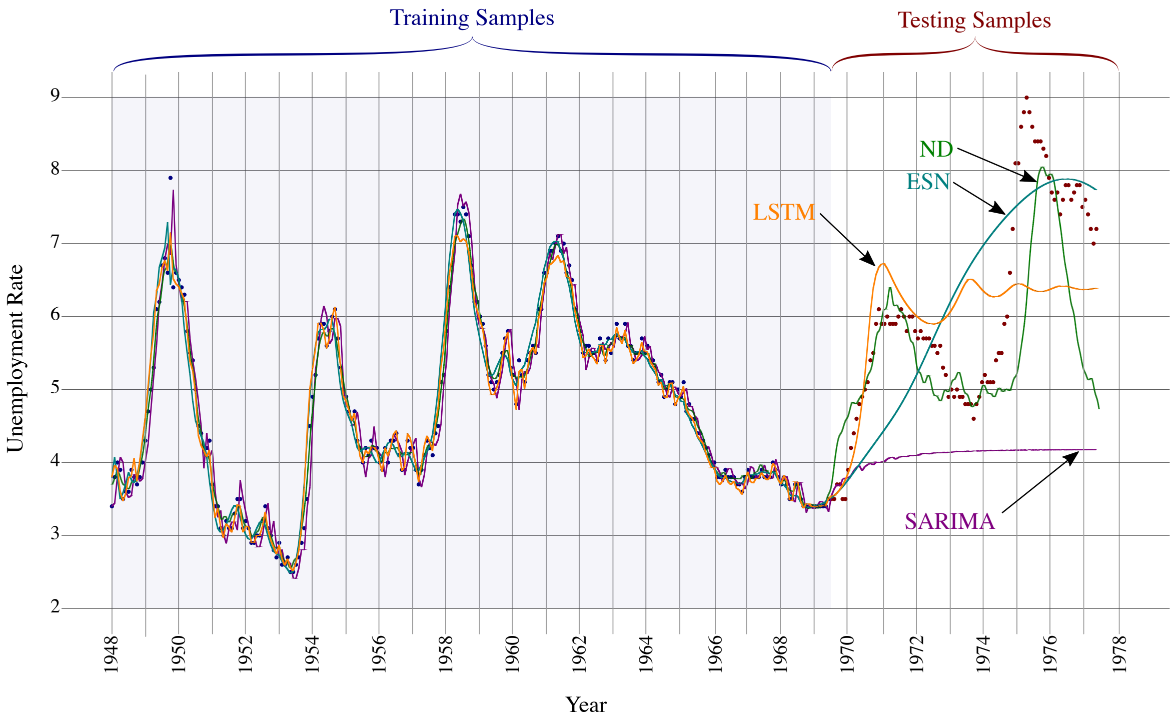

Figure 6.8: A comparison of the four best predictive models on the monthly unemployment rate in the US. Blue points represent training samples from January 1948 to June 1969 and red points represent testing samples from July 1969 to December 1977. SARIMA, shown in magenta, correctly predicted a rise in unemployment but underestimated its magnitude, and did not predict the shape of the data well. ESN, shown in cyan, predicted a reasonable mean, but did not capture the dynamics of the data. LSTM, shown in orange, predicted the first peak in the data, but leveled off to predict only the mean. Only ND, shown in green, successfully predicted both the depth and approximate shape of the surge in unemployment, followed by another surge in unemployment that followed.

Figure 6.6 demonstrates why using amplitudes provided by the Fourier transform is a poor initialization point. The actual underlying function only requires two sinusoid units (found by ND), but the Fourier transform uses every sinusoid unit available to model the linear trend in the toy problem. Instead of tuning two amplitudes, a model initialized with the Fourier transform has to tune every amplitude and is therefore far more likely to fall into local optima.

ND, by contrast, does not use the FFT. Sinusoid amplitudes (the weights feeding into the output layer) and all output-layer weights associated with $g(t)$ are initialized to small random values. This allows the neural network to learn the periodic and nonperiodic components of the signal simultaneously. Not only does this avoid unnecessary “unlearning” of the extra weights used by the DFT, but also avoids getting stuck in the local optima represented by the DFT weights. Without the hindrance of having to unlearn part of the DFT, the training process is able to find more optimal values for these weights. Figure 6.6 shows a comparison of our trained model with the frequencies used by the iDFT, omitting the linear component learned by ND.

6.3.4 Chaotic Series

In addition to the toy problem, we applied ND to the Mackey-Glass series as a proof-of-concept. This series is known to be chaotic rather than periodic, so it is an interesting test for our approach that decomposes the signal as a combination of sinusoids. Results with this data are shown in Figure 6.7. The blue points on the left represent the training sequence, and the red points on the right half represent the testing sequence. All testing samples were withheld from the model, and are only shown here to illustrate the effectiveness of the model in anticipating future samples. The green curve represents the predictions of the trained model. The series predicted by Neural Decomposition exhibits shapes similar to those in the test data, and has an RMSE of 0.086. Interestingly, neither the shapes in the test data nor those exhibited within the model are strictly repeating. This occurs because the frequencies of the sinusoidal basis functions that ND uses to represent its model may be tuned to have frequencies with no small common multiple, thus creating a signal that does not repeat for a very long time. Our model does not capture all the high-frequency fluctuations, but it is able to approximate the general shape and some of the dynamics of the chaotic series.

To determine whether Neural Decomposition merely predicts a periodic function, we tried our experiment again but set $g(t) = 0$ rather than using nonlinear, nonperiodic components for $g(t)$. We found that with these changes, our model was unable to capture the subtle dynamics of the Mackey-Glass series. As in the toy problem, omitting $g(t)$ resulted in poorer predictions, and the resulting predictions had an RMSE of 0.14 (a 63% increase in

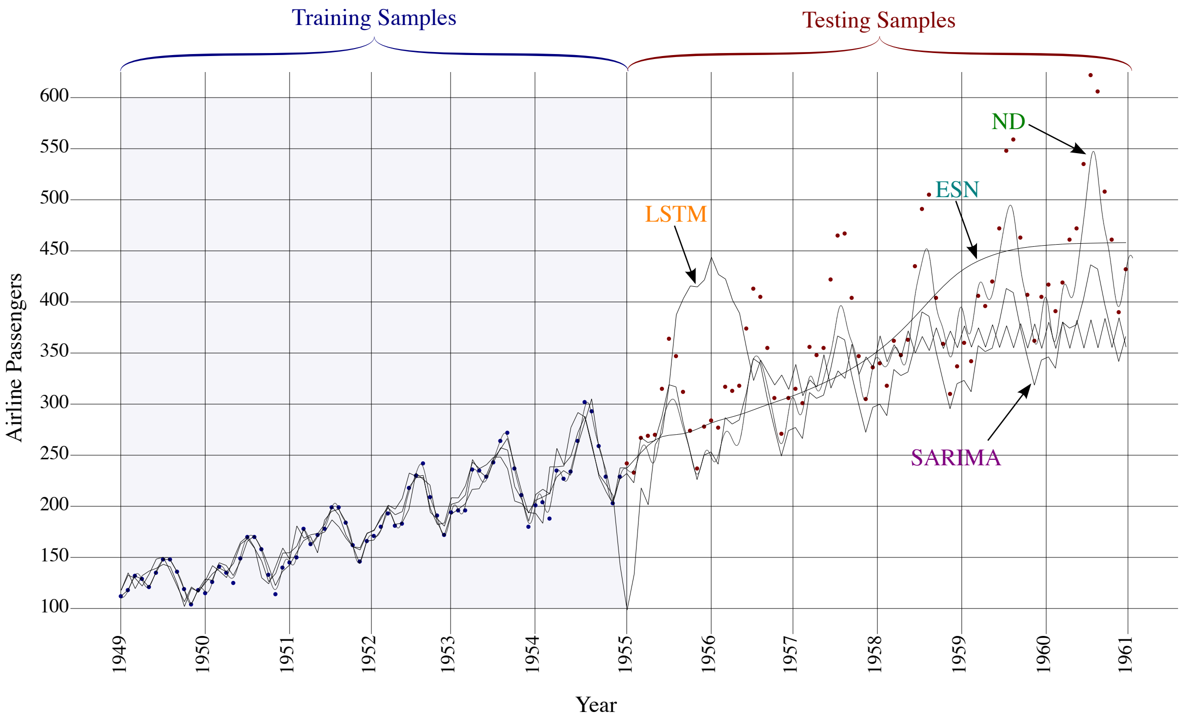

Figure 6.9: A comparison of the four best predictive models on monthly totals of international airline passengers from January 1949 to December 1960 [16]. Blue points represent the 72 training samples from January 1949 to December 1954 and red points represent the 72 testing samples from January 1955 to December 1960. SARIMA, shown in magenta, learns the trend and general shape of the data. ESN, shown in cyan, predicts a mean but does not capture the dynamics of the actual data. LSTM, shown in orange, predicts a valley and a peak that did not actually occur, followed by a poor estimation of the mean that suggests that it was unable to learn the seasonality of the data. ND, shown in green, learns the trend, shape, and growth better than the other compared models.

error). This indicates that ND does more than predict a strictly periodic function, and is able to capture at least some of the nonlinear dynamics in some chaotic systems.

Although preliminary tests on the toy problem and the Mackey-Glass series were favorable to Neural Decomposition, not all of our tests were as successful. In particular, we applied ND to another chaotic series: samples from the Lorenz-63 model. We found that ND was unable to effectively model the dynamics of this chaotic system. This seems to indicate that although ND does well with some problems, it should not be expected to anticipate all the subtle variations that occur in chaotic systems.

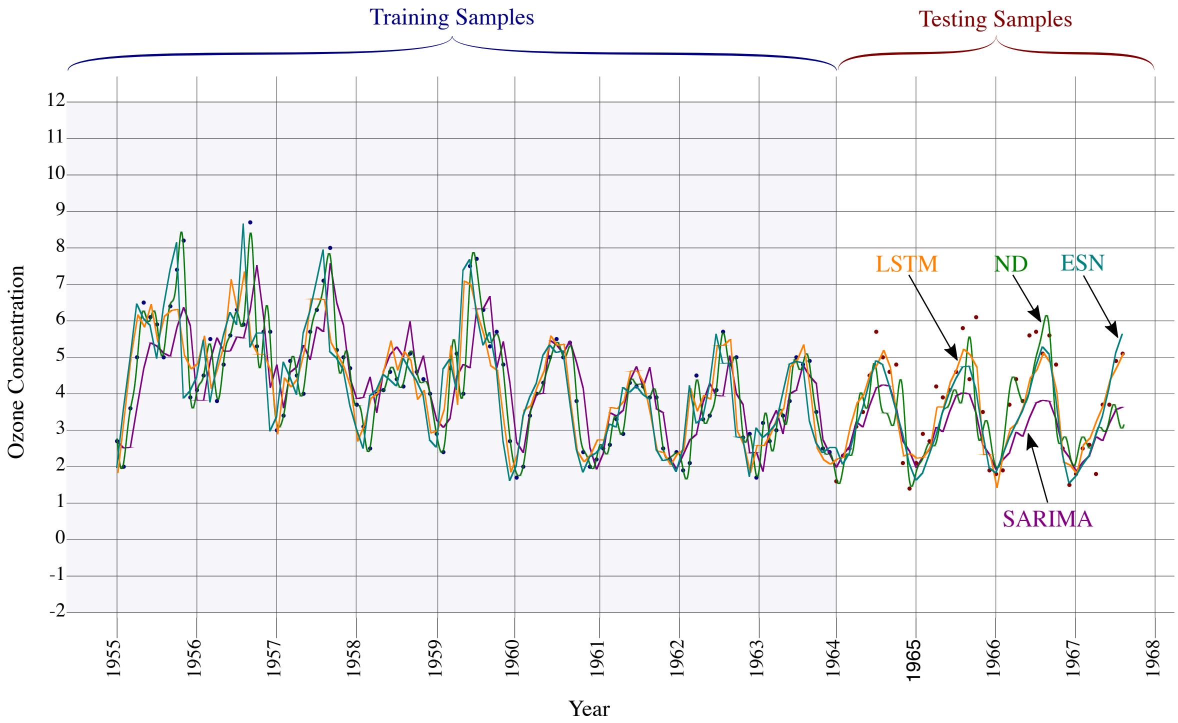

Figure 6.10: A comparison of the four best predictive models on monthly ozone concentration in downtown Los Angeles from January 1955 to August 1967 [73]. Blue points represent the 152 training samples from January 1955 to December 1963 and red points represent the 44 testing samples from January 1964 to August 1967. The compared models include SARIMA, ESN, LSTM, and ND. All four of these models perform well on this problem. Both LSTM (shown in orange) and ESN (shown in cyan) predict with slightly higher accuracy compared to ND. ND, shown in green, has slightly higher accuracy compared to SARIMA (shown in magenta). ARIMA, SVR, and Gashler and Ashmore's model all performed poorly on this problem; rather than include them in this graph, their errors have been reported in Table 6.1 and Table 6.2.

6.4 Implementation Details

In this section, we provide a more detailed explanation of our approach. A high level description of Neural Decomposition can be found in Section 6.3. For convenience, an implementation of Neural Decomposition is included in the Waffles machine learning toolkit [43].

6.4.1 Topology

We use a feedforward artificial neural network as the basis of our model. For an input of size N, the neural network is initialized with two layers: $1 \rightarrow m$ and $m \rightarrow 1$, where $m = N + |g(t)|$ and $|g(t)|$ denotes the number of nodes required by $g(t)$. The first N nodes in the hidden layer have the sinusoid activation function, $sin(t)$, and the rest of the nodes in the hidden layer have other activation functions to compute $g(t)$.

The augmentation function $g(t)$ can be made up of any number of nodes with one or more activation functions. For example, it could be made up of linear units for learning trends and sigmoidal units to fit nonperiodic, nonlinear irregularities. Gashler and Ashmore have suggested that softplus units may yield better generalizing predictions compared to standard sigmoidal units [45]. In our experiments, we used a combination of linear, softplus, and sigmoidal nodes for $g(t)$. The network tended to only use a single linear node, which may suggest that the primary benefit of the augmentation function is that it can model linear trends in the data. Softplus and sigmoidal units tended to be used very little or not at all by the network in the problems we tested, but intuitively it seems that nonlinear activation functions could be useful in some cases.

6.4.2 Weight Initialization

The weights of the neural network are initialized as follows. Let each of the N sinusoid nodes in the hidden layer, indexed as k for $0 \leq k < N$, have a weight w_k and bias ϕ_k. Let each w_k represent a frequency and be initialized to $2\pi\lfloor k/2 \rfloor$. Let each ϕ_k represent a phase shift. For each even value of k, let ϕ_k be set to $\pi/2$ to transform $sin(t + \phi_k)$ to $cos(t)$. For each odd value of k, let ϕ_k be set to π to transform $sin(t + \phi_k)$ to $-sin(t)$. A careful comparison of these initialized weights with Equation 6.1 shows that these are identical to the frequencies and phase shifts used by the iDFT, except for a missing $1/N$ term in each frequency, which is absorbed in the input preprocessing step (see Subsection 6.4.3).

All weights feeding into the output unit are set to small random values. At the beginning of training, therefore, the model will predict something like a flat line centered at zero. As training progresses, the neural network will learn how to combine the hidden layer units to fit the training data.

Weights in the hidden layer associated with the augmentation function are initialized to approximate the identity function. For example, in $g(t) = wt + b$, w is randomly perturbed from 1 and b is randomly perturbed near 0. Because the output layer will learn how to use each unit in the hidden layer, it is important that each unit be initialized in this way.

6.4.3 Input Preprocessing

Before training begins, we preprocess the input data to facilitate learning and prevent the model from falling into a local optimum. First, we normalize the time associated with each sample so that the training data lies between 0 (inclusive) and 1 (exclusive) on the time axis. If there is no explicit time, equally spaced values between 0 and 1 are assigned to each sample in order. Predicted data points will have a time value greater than or equal to 1 by this new scale. Second, we normalize the values of each input sample so that all training data is between 0 and 10 on the y axis.

This preprocessing step serves two purposes. First, it absorbs the $1/N$ term in the frequencies by transforming t into t/N, which is why we were able to omit the $1/N$ term from our frequencies in the weight initialization step. Second, and more importantly, it ensures that the data is appropriately scaled so that the neural network can learn efficiently. If the data is scaled too large on either axis, training will be slow and susceptible to local optima. If the data is scaled too small, on the other hand, the learning rate of the machine will cause training to diverge and only use linear units and low frequency sinusoids.

In some cases, it is appropriate to pass the input data through a filter. For example, financial time-series data is commonly passed through a logarithmic filter before being presented for training, and outputs from the model can then be exponentiated to obtain predictions. We use this input preprocessing method in two of our experiments where we observe an underlying exponential growth in the training data.

6.4.4 Regularization

Regularization is essential to the training process. Prior to each sample presentation, we apply regularization on the output layer of the neural network. Even though we do not initialize sinusoid amplitudes using the DFT, the network is quickly able to learn how to use the initialized frequencies to perfectly fit the input samples. Without regularizing the output layer, training halts as soon as the model fits the input samples, because the measurable error is near zero. By relaxing the learned weights, regularization allows our model to redistribute its weight over time. We find that regularization amount is especially important; too much prevented our model from learning, but too little caused our model to fall into local optima. In our experiments, setting the regularization term to 10^{-2} avoided both of these potential pitfalls.

Another important function of regularization in ND is to promote sparsity in the network, so that the redistribution of weight produces as simple a model as the input samples allow. We use L^1 regularization for this reason. Usually, the trained model does not require all N sinusoid nodes in order to generalize well, and this type of regularization enables the network to automatically discard unnecessary nodes by driving their amplitudes to zero. L^2 regularization is not an acceptable substitute in this case, as it would distribute the weights evenly throughout the network and could, like the DFT, try to use several sinusoid nodes to model what would more appropriately be modeled by a single node with a nonperiodic activation function.

It is worth noting that we only apply regularization to the output layer of the neural network. Any regularization that might occur in the hidden layer would adjust sinusoid frequencies before the output layer could learn sinusoid amplitudes. By allowing weights in the hidden layer to change without regularization, the network has the capacity to adjust frequencies but is not required to do so.

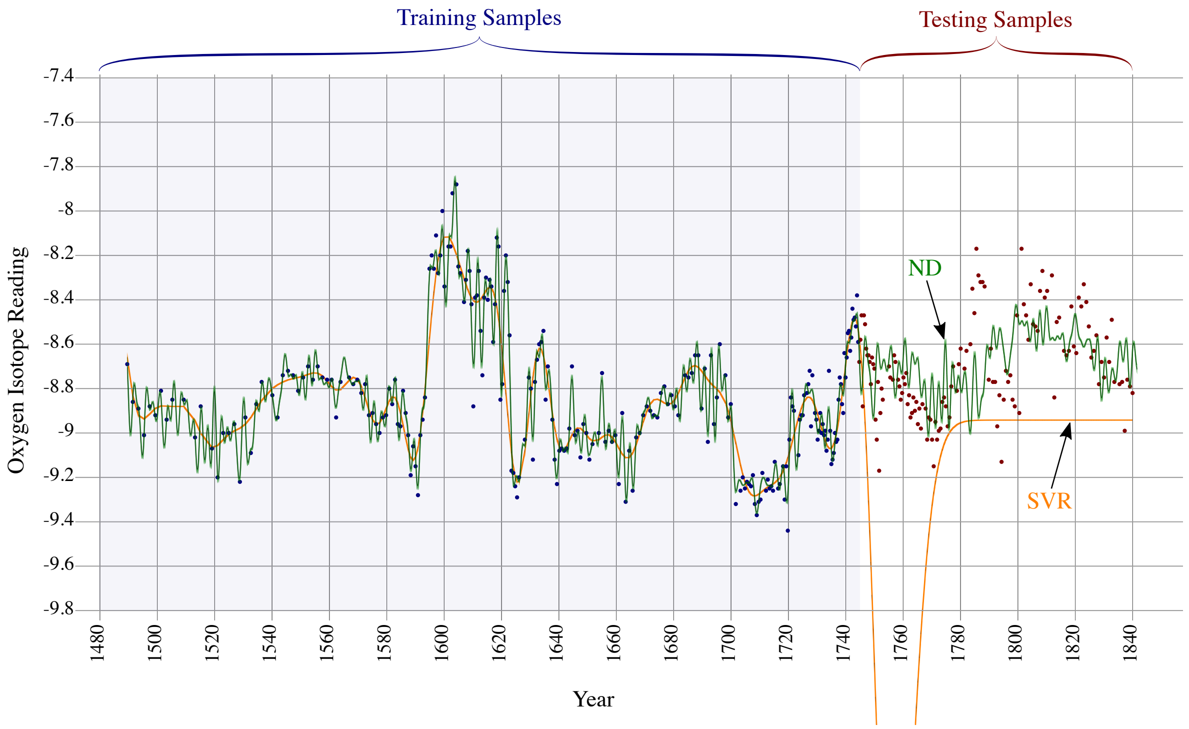

Figure 6.11: A comparison of two predictive models on a series of oxygen isotope readings in speleothems in India from 1489 AD to 1839 AD [143]. Blue points represent the 250 training samples from July 1489 to April 1744 and red points represent the 132 testing samples from August 1744 to December 1839. Because this time-series is irregularly sampled (the time step between samples is not constant), only SVR and ND could be applied to it. SVR, shown in orange, does not perform well, but predicts a steep drop in value that does not actually occur in the testing data, followed by a flat line. ND, shown in green, performs well, capturing the general shape of the testing samples.

Backpropagation with stochastic gradient descent tunes the weights of the network and accomplishes the redistribution of weights that regularization makes possible. In our experiments, we use a learning rate of 10^{-3}.

6.5 Validation

In this section, we report results that validate the effectiveness of Neural Decomposition. In each of these experiments, we used an ND model with an augmentation function made up of ten linear units, ten softplus units, and ten sigmoidal units. It is worth noting that $g(t)$ is under no constraint to consist only of these units; it could include other activation functions or only contain a single linear node to capture trend information. We use a regularization term of 10^{-2} and a learning rate of 10^{-3} in every experiment to demonstrate the robustness of our approach; we did not tune these meta-parameters for each experiment.

In our experiments, we compare ND with LSTM, ESN, ARIMA, SARIMA, SVR, Gashler and Ashmore's model [45]. We used PyBrain's implementation of LSTM networks [137] with one input neuron, one output neuron, and one hidden layer. We implemented a grid-search to find the best hidden layer size for the LSTM network for each problem and used PyBrain's RPROP- algorithm to train the network. We used Lukoševičius' implementation of ESN [114] and implemented a grid-search to find the best parameters for each problem. We used the R language implementation for ARIMA, SARIMA, and SVR [132]. For the ARIMA models, we used a variation of the `auto.arima` method that performs a grid-search to find the best parameters for each problem. For SVR, we used the `tune.svm` method, which also performs a grid-search for each problem. Although these methods select the best models based on the amount of error calculated using the training samples, the grid-search is a very slow process. Gashler and Ashmore's model did not require a grid-search for parameters because it has a default set of parameters that are automatically tuned during the training process. With ND, no problem-specific parameter tuning was performed.

In each figure, the blue points in the shaded region represent training samples and the red points represent withheld testing samples. The curves on the graph represent the predictions made by the four models that made the most accurate predictions (only two models are shown in the fourth experiment because only two models could be applied to an irregularly sampled time-series). The actual error for each model's prediction is reported for all experiments and all models in Table 6.1 and Table 6.2.

The LSTM network tended to fall into local optima, and was thus extremely sensitive to the random seed. Running the same experiment with LSTM using a different random seed yielded very different results. In each experiment, therefore, we tried the LSTM model 100 times for each different topology tested in our grid-search and selected the result with the highest accuracy to present for comparison with ND. Conversely, ND consistently made approximately identical predictions when run multiple times, regardless of the random seed.

In our first experiment, we demonstrated the effectiveness of ND on real-world data compared to widely used techniques in time-series analysis and forecasting. We trained our model on the unemployment rate from 1948 to 1969 as reported by the U.S. Bureau of Labor Statistics, and predicted the unemployment rate from 1969 to 1977. These results are shown in Figure 6.8. Blue points on the left represent the 258 training samples from January 1948 to June 1969, and red points on the right represent the 96 testing samples from July 1969 to December 1977. The four curves represent predictions made by ND (green), LSTM (orange), ESN (cyan), and SARIMA (magenta); ARIMA, SVR, and Gashler and Ashmore's model yielded poorer predictions and are therefore omitted from the figure. Grid-search found ARIMA(3,1,2) and ARIMA(1,1,2)(1,0,1)[12] for the ARIMA and SARIMA models, respectively. ARIMA, not shown, did not predict the significant rise in unemployment. SARIMA, shown in magenta, did correctly predict a rise in unemployment, but underestimated its magnitude, and did not predict the shape of the data well. SVR, not shown, correctly predicted that unemployment would rise, then fall again. However, it also underestimated the magnitude. ESN, shown in cyan, predicted a reasonable mean value for the general increase in unemployment, but failed to capture the dynamics of the actual data. The best LSTM network topology, found by grid-search, had a hidden layer with 16 neurons. LSTM, shown in orange, predicted the first peak in the data, but leveled off to predict only the mean. Gashler and Ashmore's model, not shown, predicted the rise and fall in unemployment, but underestimated its magnitude and the model's predictions significantly diverge from the subsequent testing samples. It is also worth noting that Gashler and Ashmore's model took about 200 seconds to train compared to ND, which took about 30 seconds to train.

Results with Neural Decomposition (ND) are shown in green. ND successfully predicted both the depth and approximate shape of the surge in unemployment. Furthermore, it correctly anticipated another surge in unemployment that followed. ND did a visibly better job of predicting the nonlinear trend much farther into the future.

Our second experiment demonstrates the versatility of Neural Decomposition by applying to another real-world dataset: monthly totals of international airline passengers as reported by Chatfield [16]. We use the first six years of data (72 samples) from January 1949 to December 1954 as training data, and the remaining six years of data (72 samples) from January 1955 to December 1960 as testing data. The training data is preprocessed through a $log(x)$ filter and the outputs are exponentiated to obtain the final predictions. As in the first experiment, we compare our model with LSTM, ESN, ARIMA, SARIMA, SVR, and the model proposed by Gashler and Ashmore. The predictions of the four most accurate models (ND, LSTM, ESN, and SARIMA) are shown in Figure 6.9; ARIMA, SVR, and Gashler and Ashmore's model yielded poorer predictions and are therefore omitted from the figure. SVR, not shown, predicts a flat line after the first few time steps and generalizes the worst out of the four predictive models. The ARIMA model found by grid-search was ARIMA(2,1,3). ARIMA, not shown, was able to learn the trend, but failed to capture any of the dynamics of the signal. Grid-search found ARIMA(1,0,0)(1,1,0)[12] for the SARIMA model. Both SARIMA (shown in magenta) and ND (shown in green) are able to accurately predict the shape of the future signal, but ND performs better. Unlike SARIMA, ND learns that the periodic component gets bigger over time. Gashler and Ashmore's model makes meaningful predictions for a few time steps, but appears to diverge after the first predicted season. ESN, shown in cyan, performs similarly to the ARIMA model, only predicting the trend and failing to capture seasonal variations. The LSTM network, with a hidden layer of size 64 found by grid-search, failed to capture any meaningful seasonality in the training data. Instead, LSTM immediately predicted a valley and a peak that did not actually occur, followed by a poor estimation of the mean.

The third experiment uses the monthly ozone concentration in downtown Los Angeles as reported by Hipel [73]. Nine years of monthly ozone concentrations (152 samples) from

January 1955 to December 1963 are used as training samples, and the remaining three years and eight months (44 samples) from January 1964 to August 1967 are used as testing samples. The training data, as in the second experiment, is preprocessed through a $log(x)$ filter and output is exponentiated to obtain the final predictions. Figure 6.10 compares the SARIMA, ESN, LSTM, and ND models on this problem; ARIMA, SVR, and Gashler and Ashmore's model yielded poorer predictions and are therefore omitted from the figure. The ARIMA and SARIMA models found by grid-search were ARIMA(2,1,2) and ARIMA(1,1,1)(1,0,1)[12], respectively. ARIMA and SVR resulted in flat-line predictions with a high amount of error, and Gashler and Ashmore's model diverged in training and yielded unstable predictions. SARIMA (shown in magenta), ESN (shown in cyan), LSTM (shown in orange), and ND (shown in green), on the other hand, all forecast future samples well. LSTM and ESN yielded the most accurate predictions, but SARIMA and ND yielded good results as well.

Our fourth experiment demonstrates that ND can be used on irregularly sampled time-series. We use a series of oxygen isotope readings in speleothems in a cave in India from 1489 AD to 1839 AD as reported by Sinha et. al [143]. Because the time intervals between adjacent samples is not constant (the interval is about 1.5 years on average, but fluctuates between 0.5 and 2.0 years), only ND and SVR models can be applied. ARIMA, SARIMA, Gashler and Ashmore's model, ESN, and LSTM cannot be applied to irregular time-series because they assume a constant time interval between adjacent samples; these five models are therefore not included in this experiment. Figure 6.11 shows the predictions of ND and SVR. Blue points on the left represent the 250 training samples from July 1489 to April 1744, and red points on the right represent the 132 testing samples from August 1744 to December 1839. SVR, shown in orange, predicts a steep drop in value that does not exist in the testing data. ND, shown in green, accurately predicts the general shape of the testing data.

Table 6.1 presents an empirical evaluation of each model for the four real-world experiments. We use the mean absolute percent error (MAPE) as our error metric for comparisons

Table 6.1: Mean absolute percent error (MAPE) on the validation problems for ARIMA, SARIMA, SVR, Gashler and Ashmore, ESN, LSTM, and ND. Best result (smallest error) for each problem is shown in **bold**.

Model	**Labor**	**Airline**	**Ozone**	**Speleothem**
ARIMA	39.42%	12.34%	39.50%	N/A
SARIMA	29.69%	13.33%	22.71%	N/A
SVR	25.14%	47.04%	49.53%	8.50%
Gashler/Ashmore	34.38%	19.89%	77.19%	N/A
ESN	15.73%	12.05%	**16.15%**	N/A
LSTM	14.63%	18.95%	16.52%	N/A
ND	**10.89%**	**9.52%**	21.59%	**1.89%**

[112]. MAPE for a set of predictions is defined by the following function, where x_t is the actual signal value (i.e. it is an element of the set of testing samples) and $x(t)$ is the predicted value:

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{x_t - x(t)}{x_t}\right| \tag{6.3}$$

Using MAPE, we compare Neural Decomposition to ARIMA, SARIMA, SVR with a radial basis function, Gashler and Ashmore's model, ESN, and LSTM. We found that on the unemployment rate problem (Figure 6.8), ND yielded the best model, followed by LSTM and ESN. On the airline problem (Figure 6.9), ND performed significantly better than all of the other approaches. On the ozone problem (Figure 6.10), LSTM and ESN were the best models, but ND and SARIMA also performed well. On the oxygen isotope problem (Figure 6.11), ND outperformed SVR, which was the only other model that could be applied to the irregular time-series. Table 6.1 presents the results of our experiments, and Table 6.2 presents the same data using the root mean square error (RMSE) metric instead of MAPE. In each problem, the accuracy of the best algorithm is shown in bold.

6.6 Conclusion

In this chapter, we presented Neural Decomposition, a neural network technique for time-series forecasting. Our method decomposes a set of training samples into a sum of sinusoids,

Table 6.2: Root mean square error (RMSE) on the validation problems for ARIMA, SARIMA, SVR, Gashler and Ashmore, ESN, LSTM, and ND. Best result (smallest error) for each problem is shown in **bold**.

Model	**Labor**	**Airline**	**Ozone**	**Speleothem**
ARIMA	2.97	75.32	1.33	N/A
SARIMA	2.41	67.54	1.06	N/A
SVR	2.18	209.57	1.83	1.078
Gashler/Ashmore	2.81	94.47	3.71	N/A
ESN	**1.09**	63.50	0.705	N/A
LSTM	1.14	93.61	**0.667**	N/A
ND	**1.09**	**45.03**	0.99	**0.214**

inspired by the Fourier transform, augmented with additional components to enable our model to generalize and extrapolate beyond the input set. Each component of the resulting signal is trained, so that it can find a simpler set of constituent signals. ND uses careful initialization, input preprocessing, and regularization to facilitate the training process. A toy problem was presented to demonstrate the necessity of each component of ND. We applied ND to the Mackey-Glass series and was found to generalize well. Finally, we showed results that demonstrate that our approach is superior to popular techniques LSTM, ESN, ARIMA, SARIMA, SVR, and Gashler and Ashmore's model in some cases, including the US unemployment rate, monthly airline passengers, and an unevenly sampled time-series of oxygen isotope measurements from a cave in north India. We also showed that in some cases, our approach is at least comparible to these other techniques, as in the monthly time-series of ozone concentration in Los Angeles. We predict that ND will similarly perform well on a number of other problems.

This work makes the following contributions to the current knowledge:

- It empirically shows why the Fourier transform provides a poor initialization point for generalization and how neural network weights must be tuned to properly decompose a signal into its constituent parts.

- It demonstrates the necessity of an augmentation function in Fourier and Fourier-like neural networks and shows that components must be adjustable during the training process, observing the relationships between weight initialization, input preprocessing, and regularization in this context.

- It unifies these insights to describe a method for time-series forecasting and demonstrates that this method is effective at generalizing for some real-world datasets.

The primary area of future work is to apply ND to new problems. The preliminary findings on the datasets in this chapter show that ND can generalize well for some problems, but the breadth of applications for ND not yet known. Some interesting areas to explore are traffic flow [112], sales [23], financial [155], and economic [90].

Chapter 7

Leveraging Product as an Activation Function

Abstract: Product unit neural networks (PUNNs) are powerful representational models with a strong theoretical basis, but have proven to be difficult to train with gradient-based optimizers. We present windowed product unit neural networks (WPUNNs), a simple method of leveraging product as a nonlinearity in a neural network. Windowing the product tames the complex gradient surface and enables WPUNNs to learn effectively, solving the problems faced by PUNNs. WPUNNs use product layers between traditional sum layers, capturing the representational power of product units and using the product itself as a nonlinearity. We find the result that this method works as well as traditional nonlinearities like ReLU on the MNIST dataset. We demonstrate that WPUNNs can also generalize gated units in recurrent neural networks, yielding results comparable to LSTM networks.

7.1 Introduction

Nodes in an artificial neural network traditionally apply a nonlinearity to a weighted sum of its inputs. Previous work suggests that using a product rather than a sum should increase the representational power of a neural network, but these product unit neural networks (PUNNs) have proven difficult to train using gradient-based optimizers [38]. Given an input of N elements, a product unit multiplies together all N elements raised to arbitrary powers (where each exponent is a learned parameter). It is not surprising, then, that gradient descent has difficulty training networks of product units; the derivative of an N element product with respect to one of its elements is an $N-1$ element product. Thus, the error surface of a PUNN is particularly chaotic and contains many poor local optima.

We present windowed product unit neural networks (WPUNNs), a simple method of leveraging product as a nonlinearity in a neural network. Windowing the product tames the complex gradient surface and enables WPUNNs to learn effectively, solving the problems faced by PUNNs. A windowed product unit takes the product not of all N inputs, but on a small portion of the inputs (a window), significantly reducing gradient complexity. WPUNNs use layers of these windowed product units between traditional sum layers, capturing the representational power of product units and using the product itself as a nonlinearity. In this chapter, we make three discoveries related to WPUNNs:

- gradient-based optimization can be used effectively with windowed units,
- windowed product is as effective as traditional nonlinearities like rectified linear units (ReLU), and
- WPUNNs can generalize gated units in recurrent neural networks.

The rest of this chapter is laid out as follows. Related work is briefly discussed in Section 7.2. In Section 7.3, we present the formulation of WPUNNs. Section 7.4 lays out a couple of theoretical applications of WPUNNs. We present our findings in Section 7.5 and offer some concluding thoughts in Section 7.6.

7.2 Related Work

7.2.1 Product Unit Neural Networks

Product unit neural networks (PUNNs) were introduced in the late 1980s as computationally powerful alternatives to traditional summation unit networks [36]. In a PUNN, weights are used as exponents rather than as coefficients, giving a single product layer followed by a sum enough representational power to represent arbitrary polynomials. The standard forumulation of a PUNN unit is

$$y = \Pi_{i=1}^{N} x_i^{\theta_i}, \tag{7.1}$$

where y is the output, N is the number of inputs, x_i is the ith input, and θ_i is the ith weight. The use of product instead of sum increases a network's information capacity [87, 163] and enables higher-order combinations of inputs [38]. The two drawbacks to this approach are the overhead of computing so many exponents and logarithms [87] and the difficulty of training a PUNN with gradient-based optimization methods [38]. This second problem has proven to be particularly challenging, so most of the work on PUNNs focus on alternative, non-gradient-based techniques such as genetic and evolutionary algorithms [87, 38, 79, 71, 72, 119] and particle swarm optimizers [38, 79, 161]. Despite the difficulty in training them, PUNNs have been applied to a number of problems, including classification of Listeria growth [119], massive missing data reconstruction [35], and time series forecasting [41]. More recent work has focused on hybridizing PUNNs with sigmoid and RBF networks [62, 64, 63, 133], and developing improved evolutionary algorithms for training PUNNs [152, 153, 60]. Our work discovers a way to capture the representational power of a PUNN without losing the ability to train the network using standard gradient-based optimization algorithms.

7.2.2 Gated Units

Recurrent neural networks often make use of a product to implement gates that control the flow of information in a sequence. Long short term memory (LSTM) [75, 48] and gated recurrent unit (GRU) [22, 25] networks use a combination of sigmoids and products to "gate" information. Gated units use the input and recurrent connections to compute several gate values (in the range [0..1]) and a memory value. Some gated units use one of the gate values as an input gate, controlling how much of the input is used when calculating the new memory value. Some have a forget gate, controlling how much of the previous memory value is used when calculating the new memory value. Still others have an output gate, controlling how much of the memory value is actually used as output. Gates and outputs are generally computed using a standard feedforward approach (a nonlinearity applied to a weighted sum of inputs), then multiplied in a pre-defined way depending on which model is used (for example, LSTM or GRU). LSTM networks, in particular, have proven to be particularly powerful models for speech recognition [58], language modeling [47], text-to-speech synthesis [40], and handwriting recognition and generation [59, 57]. Our approach is related to gated units by the use of multiplication and can be used as a generalization of gated units.

7.2.3 Sum-Product Networks

The use of the product operation has been shown to be effective in graphical inference models. In 2011, Poon et. al proposed a deep architecture for graphical inference models called Sum-Product Networks (SPN). This architecture not only has a strong mathematical basis, but also yields compelling results, surpassing state-of-the-art deep neural networks at the image completion task [129]. A SPN is arranged into layers, as in a multilayer perceptron. Inputs to a SPN are binary (either 0 or 1), and layers alternate between weighted sums and weighted products (where the weights used in sums are in [0..1] and the weights used in products are either 0 or 1). SPNs have been successfully used for classifying videos of various activities [3, 4], bandwidth extension of speech signals [127], image classification for autonomous flight [139], and more [107, 100, 120]. Our work borrows the idea of alternating sum and product as used in SPN graphical models. Although neural networks have much in common with graphical models, they are better-suited for learning in domains where little is known *a priori* about the relations among available attributes.

7.3 Approach

In our approach, we use a specialized kind of product unit in a neural network. Our method has two key differences from the standard product unit used in PUNNs. First, our product is weightless; we do not use weights as exponents in the product. Second, our product is windowed; rather than having each unit take a product of all elements in the input vector, each unit takes a product of a small window of the inputs. WPUNNs have two hyperparameters: the window size w, $1 \leq w \leq N$, and stride length s, $1 \leq s \leq w$. Given a vector of input values, we slide a window of size w using a stride of s and multiply all elements in the window to yield an output value. In the simplest case, where $w = 2$ and $s = 2$, every odd input is multiplied by its adjacent even input and the input vector is reduced by a factor of 2. Formally, for an input vector x of size N, the output is a vector y of size $M = (N - w + s - 1)/s + 1$ as defined by the following equation, for $1 \leq i \leq M$:

$$y_i = \Pi_{j=0}^{w-1} x_{si+j}. \tag{7.2}$$

Note that changing the product to the *max* aggregation function changes the operation into max pooling, which is commonly used in convolutional neural networks [102].

By inserting a layer of these windowed product units between each fully-connected layer in a neural network, we build a windowed product unit neural network (WPUNN). We make the following observations about a WPUNN unit:

- It is differentiable; $\frac{dy_i}{dx_j} = y_i/x_j$.
- It is a generalization of PUNN units with respect to window size; a WPUNN unit with $w = N$ is equivalent to a PUNN unit.
- It is a specialization of PUNN units with respect to weights; a PUNN unit where all weights are 1 is a WPUNN unit.
- Its derivative is less chaotic than the derivative of a PUNN unit; all exponents are 1, and (generally) $w < N$.
- It is nonlinear with respect to each input, because it is multiplied by a variable and not a scalar.

These observations yield a number of corollaries. First, because WPUNN derivitives are less chaotic than PUNN derivatives, WPUNNs are able to learn effectively without any other explicit nonlinearity (such as rectified linear or sigmoidal units). With a small

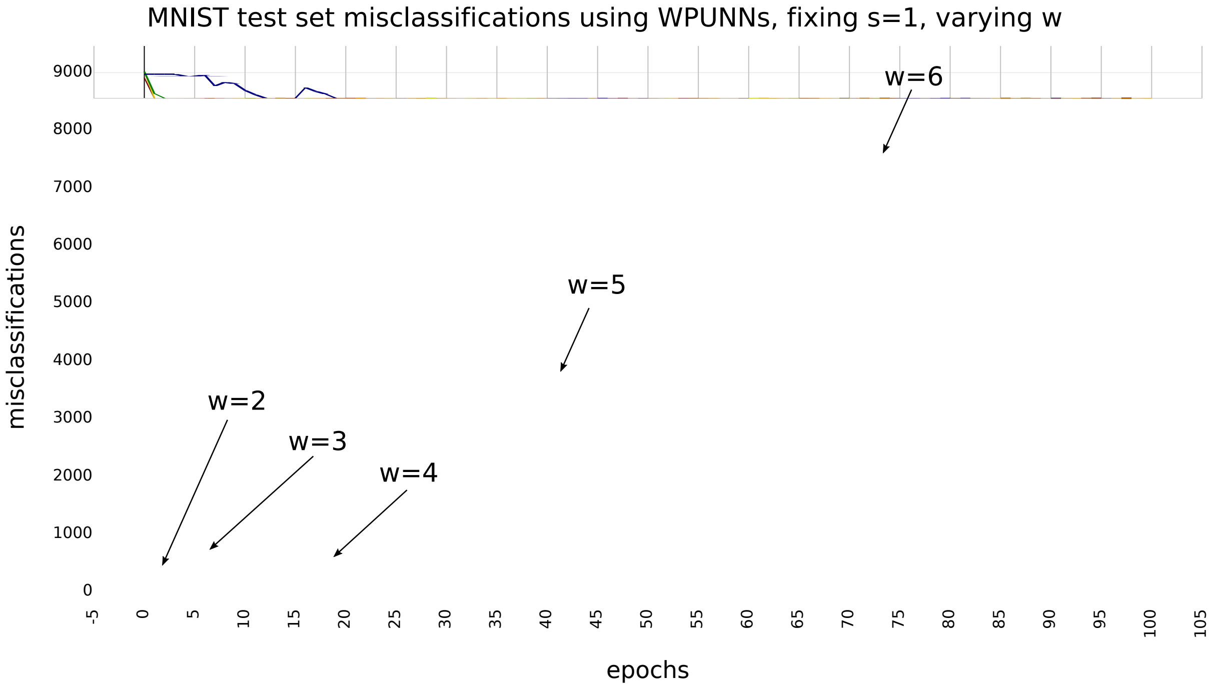

Figure 7.1: Test set misclassifications over time of a WPUNN on the MNIST dataset, fixing $s = 1$ but varying w. The inverse relationship between accuracy and window size demonstrates that WPUNNs are sensitive to the hyperparameter w (window size). Reducing window size solves the major problems associated with training PUNNs, which validates our primary contribution.

enough window size, such a network should not experience the same training problems as experienced in training PUNNs, which should enable it to learn using standard gradient-based optimization techniques rather than resorting to evolutionary algorithms. Second, because WPUNN layers are weightless, the forward and backward propagation steps are more efficient than in PUNNs. Third, because a WPUNN with a high w is more like a PUNN, WPUNNs are sensitive to this hyperparameter and will favor lower values of w. Fourth, because a fully-connected layer preceeds all WPUNN units, WPUNNs are not sensitive to the stride length hyperparameter s. With a single fully-connected layer, a neural network can weight, rearrange, and duplicate inputs. Thus, a WPUNN layer after a sufficiently wide fully-connected layer does not need stride to overlap window, and a WPUNN should be as effective with $s = 1$ as with $s = w$.

7.4 Applications

7.4.1 Representing Polynomials

A WPUNN can exactly represent arbitrary polynomials. Although this appears to be a simple problem, neural networks with standard nonlinearities such as *tanh* and *ReLU* can only, at best, approximate these values. Representing polynomials is particularly useful when modeling dynamical systems, which are often derived from polynomial equations [88].

WPUNNs can model any monomial, x^d, as a network with a one input and one hidden layer. The input is simply x. The hidden layer is a fully-connected layer of width d, with all weights set to 1 and all biases set to 0. This effectively duplicates the input d times. Finally, the output layer is a WPUNN layer with window size $w = d$ (i.e. a single product). To extend this to polynomials, we can design a network that repeats this construction for each monomial in the polynomial, using a window size corresponding to the monomial with the highest degree and a stride of the same size.

WPUNNs are not unique in their ability to represent polynomials. In fact, PUNNs can exactly represent arbitrary polynomials in a single *unit* rather than in two layers, as weights in PUNNs are exponents. However, PUNNs have proven to be difficult to train with traditional gradient-based techniques, while WPUNNs do not have this problem.

7.4.2 Generalizing Gated Units

WPUNNs can generalize gated units. Gated units depend on the multiplication of inputs, outputs, and hidden states ("cell value" in LSTM) with their corresponding gate values. Because these values are computed as dot products, the gating operation is equivalent to a WPUNN layer with $w = s = 2$. Thus, we can design a WPUNN-based gated unit architecture that similarly controls the flow of information.

One possible architecture for gated units using WPUNNs is the following 3-layer block, where N is the number of outputs. The first layer is a fully-connected layer that takes x_t (the

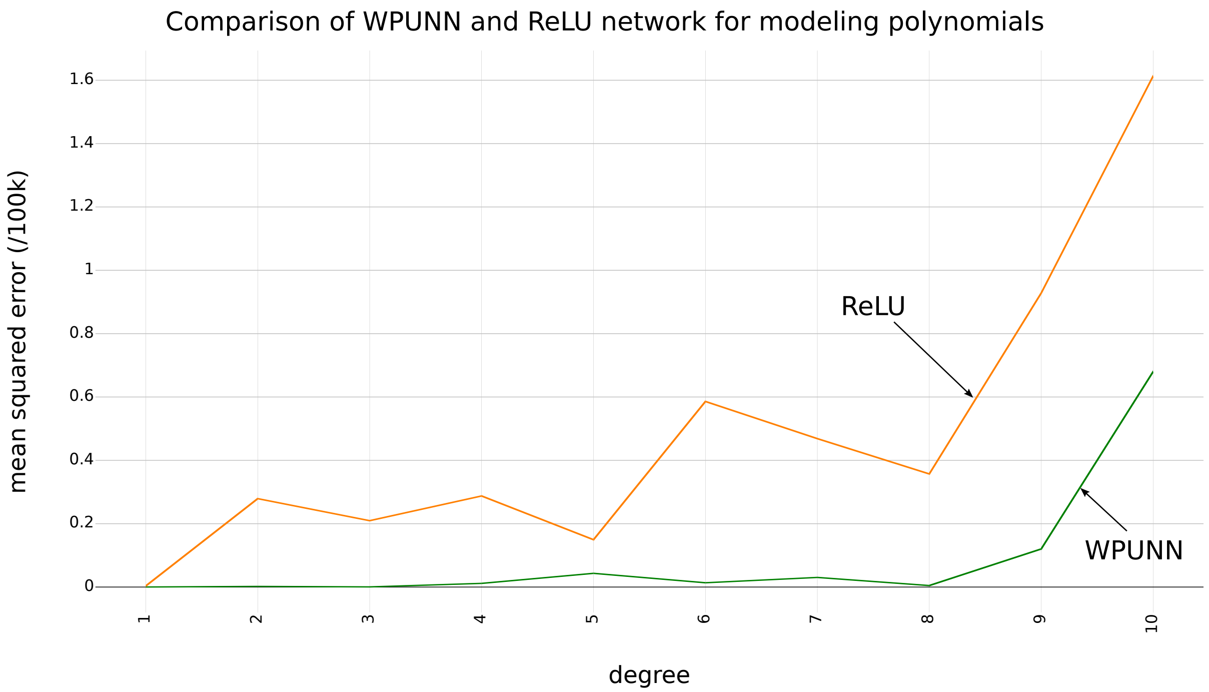

Figure 7.2: A comparison of two neural networks for modeling polynomials. The vertical axis is error (lower is better) and the horizontal axis is polynomial degree. The green curve is the error rate from a WPUNN and the orange curve is the error rate from a ReLU network. As expected, the WPUNN consistently yields a lower error rate than the ReLU model. The noticable increase in loss in the WPUNN for $d = 9$ and $d = 10$ can be attributed to the depth of the network being less than $log(d)$.

current input) and y_{t-1} (the output at the previous time step) as inputs and produces $2N$ values. This allows the network to rearrange the inputs as needed; if we assume that existing RNN units are optimal, this will interleave elements of x_t with "input gate values" and the elements of y_{t-1} with "output gate values". The second layer is a sigmoid that flattens the output from the first layer into the range $[0..1]$. Rather than using one nonlinearity for squashing the input and output and another for squashing the gates, we use the same nonlinearity for both. The final layer is a windowed product layer with $w = s = 2$. This layer effectively gates the odd-indexed units by the even-indexed units (or vice versa) and reduces the $2N$ values from the first two layers to N values, which is the target number of outputs. We present a comparison of this proposed architecture with long short term memory (LSTM) networks in Section 7.5.

7.5 Results

In our first two tests, we use the MNIST dataset to evaluate our claims that WPUNNs are insensitive to the hyperparameter s (stride length) and that they are sensitive to the hyperparameter w (window size). In both tests, we use a WPUNN with the following 6-layer topology: 1) a fully-connected layer to 300, 2) a WPUNN layer (parameters s and w vary), 3) a fully-connected layer to 100, 4) a WPUNN layer (parameters s and w vary), 5) a fully-connected layer to 10, and 6) a log soft max layer. The network is trained to minimize negative log loss using the Adam optimizer with a learning rate of $1e^{-4}$.

For the first experiment, to test our claim that WPUNNs are insensitive to the hyperparameter s (stride length), we try varing the hyperparameters while training on MNIST. For this test, we fix $w = 4$ and vary s from 1 to 4. If WPUNNs are not sensitive to the choice of s, we would expect to see a similar learning curve for these four models. All four models we tested performed with near-identical success, with an average test set misclassification rate of 3.56% and a variance of 0.29. This supports our theory that WPUNNs are not sensitive to the hyperparameter s.

For the second experiment, to test our claim that WPUNNs are sensitive to the hyperparameter w (window size), we fix $s = 1$ and vary w from 2 to 8. If WPUNNs are sensitive to w, we would expect to see a higher learning accuracy on the models near $w = 2$ and a lower learning accuracy on the models near $w = 8$. Figure 7.3 shows a plot of the test set error over time for these models. As expected, there is an inverse correlation between w and the accuracy, degrading dramatically when $w > 4$. This supports our theory that WPUNNs are sensitive to w, and $w = 2$ yields higher accuracy. Thus, reducing window size solves the major problems associated with training PUNNs, which validates our primary contribution. As explained in Section 7.3, fixing $w = s = 2$ does not significantly affect the representational power of a WPUNN that is sufficiently deep (logarithmic with respect to degree) and wide (at most quadratic with respect to the number of inputs).

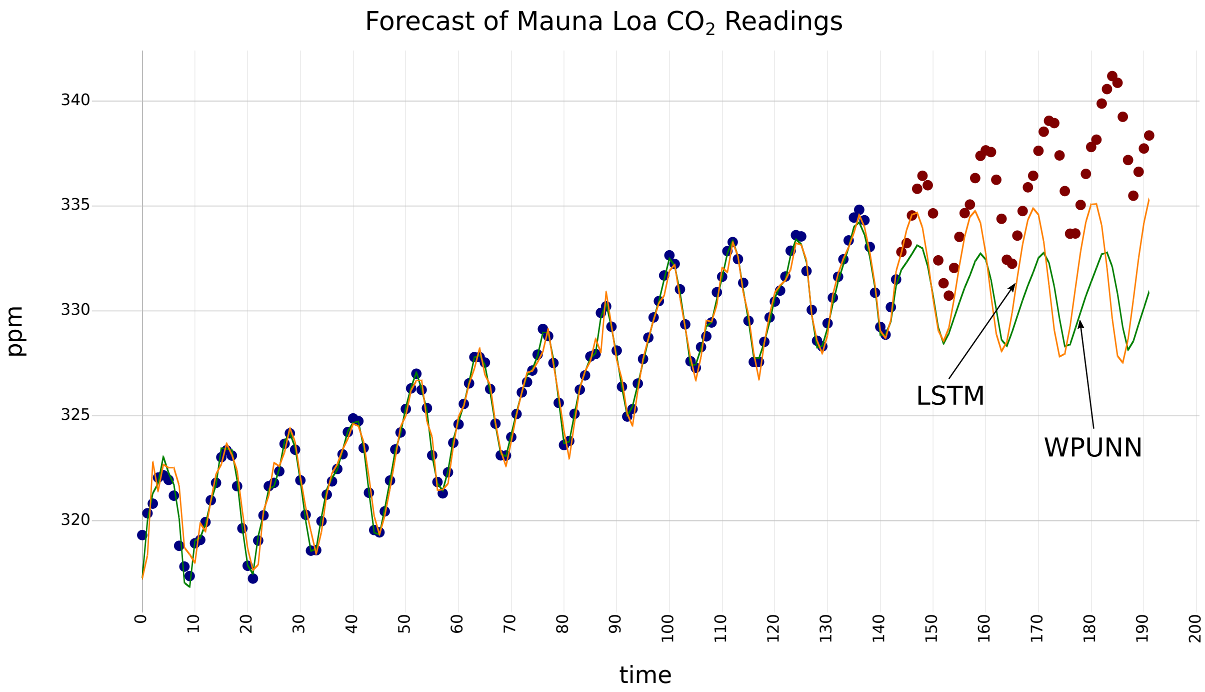

Figure 7.3: Forecasts made by two models on a time-series of Mauna Loa CO2 readings. Blue points represent training data, red points represent withheld testing data, the green curve represents the forecast by a WPUNN, and the orange curve represents the forecast by a LSTM network. Although the LSTM network yields a more accurate prediction, the WPUNN model is faster and uses a simpler architecture.

In our third experiment, we test how well a WPUNN can represent polynomials by generating a random polynomial, sampling training data from that polynomial, then testing how accurately a trained WPUNN can calculate the polynomial value given a new set of values for the variables. We generate a polynomial with two variables, x and y, and one term for each monomial of degree d or less (varying d in our experiments). The coefficients on the terms are drawn from a uniform distribution in $[-1, 1]$. We then generate 1000 training samples by selecting values for x and y also drawn from a uniform distribution in $[-1, 1]$, using the values of x and y as features and the computed values of the polynomial as labels. We repeat this generation to obtain 1000 testing samples that are withheld from the model. We compare two models with similar topologies. The first model is a WPUNN with the following topology: 1) a fully-connected layer to 50, 2) a WPUNN layer, 3) a fully-connected layer to 50, 4) a WPUNN layer, 5) a fully-connected layer to 50, 6) a WPUNN layer, and 7) a fully-connected layer to 1. The second model is identical except that, instead of WPUNN, we use a leaky ReLU activation with parameter $\ell = 0.1$. With this construction, the WPUNN model has 2776 parameters while the ReLU model has 5301 parameters, and the WPUNN trains approximately twice as fast as the ReLU network. The networks are trained to minimize mean squared error using the Adam optimizer with a learning rate of $1e^{-3}$. We repeat the experiment varying the degree of the polynomial from $d = 1$ to $d = 10$ and plot to results in Figure 7.4.2. As expected, the WPUNN consistently yields a lower error rate than the ReLU model. The noticable increase in loss in the WPUNN for $d = 9$ and $d = 10$ can be attributed to the depth of the network being less than $log(d)$.

In our fourth experiment, we test how well the gated unit construction proposed in Section 7.4.2 works. We compare a WPUNN with a LSTM network on the Mauna Loa CO2 ppm time series [74]. The WPUNN uses the following topology: 1) a fully-connected layer to 100, 2) a sigmoid activation, 3) a WPUNN layer with a recurrent connection to layer 1, 4) a fully-connected layer to 100, 5) a sigmoid activation, 6) a WPUNN layer with a recurrent connection to layer 4, 7) a fully-connected layer to 1. The LSTM network uses the following topology: 1) a LSTM layer to 100, 2) a LSTM layer to 100, 3) a fully-connected layer to 1. Despite the difference in depth (7-layer WPUNN as opposed to a 3-layer LSTM network), the WPUNN has 9026 parameters while the LSTM network has 27401 parameters, and the WPUNN trains approximately three times as fast as the LSTM network. We withhold the last 25% of the CO2 time-series data for testing and train both models on the first 75%. The networks are trained to minimize mean squared error using the Adam optimizer with a learning rate of $1e^{-2}$. A plot of the forecasts from each model is shown in Figure 7.5. The blue points represent the training data. The red points represent the withheld testing data. The green curve is the forecast from the WPUNN, and the orange curve is the forecast from the LSTM network. The mean squared error for the WPUNN forecast is 0.12, while the mean squared error for the LSTM forecast is 0.085. The LSTM network outperforms the WPUNN model, but the WPUNN model performs notably well despite using only a third of the weights and a significantly simpler architecture. It is worth noting that because WPUNN generalizes LSTM, if we used a more complex WPUNN architecture, the WPUNN predictions would be as accurate as the LSTM predictions.

7.6 Conclusion

The results presented in Section 7.5 demonstrate that WPUNNs are effective machine learning models. We make two primary contributions: 1) we give a construction of PUNNs that can be trained by gradient-based optimizers, and 2) we provide empirical results demonstrating the utility of using product as a nonlinearity in a neural network. Without any other nonlinearity, WPUNNs work as well as traditional neural networks using fewer weights. WPUNNs can be used to construct gated units in recurrent neural networks to model time-series data as effectively as LSTM networks.

Chapter 8

An Evaluation of Parameterized Activation Functions for Deep Learning

Abstract: Parametric activation functions, such as PReLU and PELU, are a relatively new subdomain of neural network nonlinearities. In this chapter, we present a full comparison of these methods across several topologies. We find that parameterizing activation functions in neural networks do not tend to overfit and tend to converge more quickly than non-parametric activations. We also introduce the Bendable Linear Unit (BLU), which synthesizes many useful properties of other activations, including PReLU, ELU, and SELU. Our experiments indicate that parametric activations that can approximate the identity function are more robust against losing residual connections in deep networks. On the CIFAR-10 task, BLU outperforms other activations when using a topology without residual connections.

8.1 Introduction

Nonlinear activation functions enable neural networks to fit complex data in high dimensional space. For decades, sigmoidal nonlinearities were the most prevalent activation functions [91]. More recently, rectified linear units (ReLUs) [125] have risen in popularity, helping to solve the long-standing problem of vanishing gradients. A number of variations of ReLUs have been proposed, including leaky ReLUs [115], randomized ReLUs [165], and more. Parameterized or parametric activation functions, such as parametric ReLU [67], are a relatively new subdomain of neural network nonlinearities. In some cases, neural networks that employ these newer activation functions tend to have greater representational power, converge faster, and yield higher accuracies than traditional sigmoidal neural networks [52, 67, 158].

Although earlier activations included fixed hyperparameters (such as the “leak” in leaky ReLUs or the negative asymptote in exponential linear units [27]), parameterized activation functions are unique in that their parameters are learned as network weights during training.

This kind of nonlinearity is more flexible and can be fine-tuned to produce a more accurate model than networks using non-parametric activations. In 2015, parametric ReLUs were demonstrated to outperform traditional ReLUs on the ImageNet task [67], and in 2017, parametric ELUs were similarly demonstrated to be superior to non-parametric ELUs on the MNIST, CIFAR-10/100, and ImageNet tasks [158].

Despite promising initial results, little work has been done to determine whether these parametrizations offer any significant advantage over their non-parametric counterparts. Out of the few papers that compare PReLU with ReLU, some have demonstrated that PReLU performs better [67, 169], while others have found PReLU to yield results that are about the same [165, 180]. Other paramtric activations have almost no work comparing them to standard nonlinearities. Thus, it is not yet known whether the parameters introduced by such functions carry any more representational value than other network parameters.

In this chapter, we address the topic of parametric activation functions and seek to answer the question, *is the parameterization of activation functions beneficial in terms of increased accuracy or lower training times?* To answer this question, we compare a number of parametric and non-parametric activations and observe the relationship between a network's nonlinearity, width, depth, and final predictive accuracy. We also introduce a new parametric activation function, the Bendable Linear Unit (BLU), providing a theoretical basis and compelling empirical results. Our experiments demonstrate that

- parametric activations achieve (marginally) higher accuracy than their non-parametric counterparts,
- parametric activations tend to converge more quickly than non-parametric activations, and
- parametric activations that can approximate the identity function are more robust than non-parametric activations when removing residual connections.

Thus, parameterizing activation functions is generally beneficial, enhancing the representational power of the network without a great risk of overfitting and speeding up convergence. The remainder of this chapter is split into four sections. Section 8.2 discusses related work on parameterized activation functions. Section 8.3 introduces BLU a provides a theoretical basis for its use. Section 8.4 presents our experimental setup and results, and in Section 8.5 we make some conclusing remarks.

8.2 Related Work

8.2.1 Adaptive Transfer Functions

Adaptive transfer functions, another term for parametric activation functions, were proposed as early as 2001 [34]. They were initially framed as enabling smaller networks to be more precise by means of flexible decision borders. Proposed earlier than ReLUs, these early parametric methods included adaptive step functions, sigmoidal functions, radial basis functions, Gaussian functions, and more. At the time, however, they served as little more than theoretical curiosities, and were not applied to any tasks.

8.2.2 Adaptive Piecewise Linear Units

In 2014, a parametric activation function called adaptive piecewise linear unit (APLU) was proposed [2]. An APLU network seeks to mimic the highly successful maxout network [55, 6, 124] without as many extra parameters. Compared to a network using a standard non-parametric activation function, a maxout network increases layer widths by a constant factor, resulting in an increase in weights that is quadratic in the number of hidden units in the network. The parameter increase for APLU networks, on the other hand, is linear in the number of hidden units. An APLU is a piecewise function made up of S linear components, where S is a pre-selected hyperparameter. Each component has a slope, a_i, and a bias b_i. The biases also control the "hinges" of an APLU, determining where the pieces connect. Thus, a layer of APLUs add $2SN$ parameters to the network where N is the number of units in the layer. The output of a single APLU is

$$h(x) = max(0, x) + \sum_{i=1}^{S} a_i max(0, x + b_i). \tag{8.1}$$

APLU networks achieved state-of-the-art accuracy on the CIFAR-10 and CIFAR-100 datasets [101] at the time they were introduced. It is noteworthy that APLUs were able to improve accuracy over maxout networks despite the fact that APLUs use only a fraction of the parameters used by a maxout network.

8.2.3 Parametric Rectified Linear Units

A traditional ReLU is a simple piecewise function: if the input, x, is negative, the output is 0; if x is positive, the output is x itself [125]. Thus, a ReLU has a gradient of 1 for positive inputs and 0 for negative inputs. This has the undesirable consequence of causing training to slow or halt for units that receive mostly negative inputs. Weights feeding into units with a 0 gradient stop training, so portions of the network effectively "die" during training. Leaky ReLUs (LReLUs) avoid this problem by multiplying negative inputs by a small value (the "leak") rather than thresholding them to 0. This small change improves accuracy over standard ReLUs [115] by preventing units from having a 0 gradient.

Parametric ReLUs (PReLUs) were introduced in 2015 as a dynamic version of Leaky ReLUs (LReLUs) [67]. PReLUs allow specialization in the network by adaptively learning the "leak" parameter for each unit during training. For the ImageNet task [135], networks using PReLU as their nonlinearity have been shown to yield better results than ReLU and LReLU networks [67]. However, for the CIFAR-10 and CIFAR-100 [101] tasks, PReLU was shown to be susceptible to overfitting problems to the extent that, for those two tasks, nonparametric LReLU produced higher classification accuracy [165]. A PReLU is simpler than the earlier-proposed APLU, but it has received more attention and has produced superior results.

8.2.4 Parametric Exponential Linear Units

Exponential linear units (ELUs) were proposed in 2015 as an enhancement to ReLUs [27]. ELUs are similar to LReLUs, with an important difference: rather than using a linear component for negative values, an exponential term, $e^x - 1$ is used. The major effect of this adjustment is that the range of the activation is changed from $[-\inf, \inf]$ to $[-1, \inf]$, such that networks using this activation have a self-regularizing tendency to prefer values that are either positive or have a small magnitude. As a result, ELUs tend to outperform ReLUs, with or without the addition of explicit regularization methods such as Dropout [145] and Batch Normalization [78]. ELUs have a fixed hyperparameter, α, which controls the negative asymptote such that the range of an ELU is $[-\alpha, \inf]$, although the authors recommended using a fixed value of $\alpha = 1$.

Just as PReLUs generalize LReLUs, Parametric Exponential Linear Units (PELUs) generalize ELUs by allowing the parameter, α, to be learned dynamically during the training process [158]. PeLUs actually have two learnable parameters: α, the negative asymptote, and β, which adjusts the slope of the positive side of the output. The output of a PELU is characterized by the following piecewise equation:

$$\begin{cases} \frac{\alpha}{\beta} h & x \geq 0 \\ \alpha(e^{\frac{h}{\beta}} - 1) & x < 0. \end{cases}$$

Both α and β are learned as network parameters in a PELU network. PELUs have been shown to outperform ELUs on the CIFAR-10, CIFAR-100, and ImageNet tasks [158]. In addition to having a better final accuracy, PELUs were reported to yield a more stable error curve throughout the training process compared to non-parameterized ELUs.

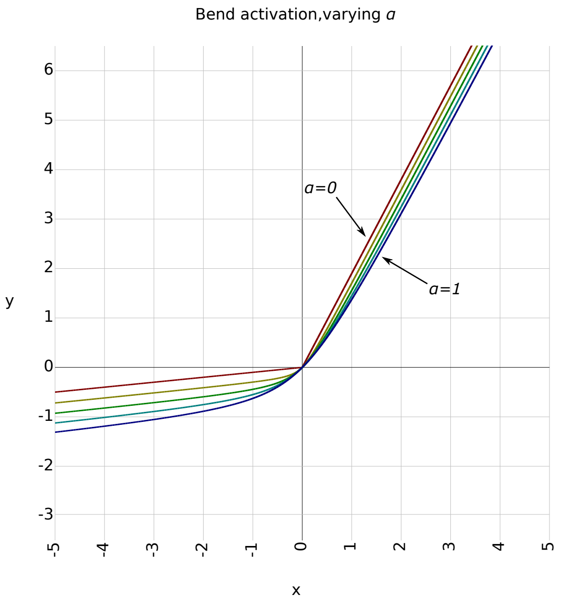

Figure 8.1: A visualization of Bendable Linear Units, varying α, with β fixed at 0.9. The closer α is to 0, the more it is like LReLU [115], and the closer α is to 1, the more it is like SoftPlus [51].

8.3 Bendable Linear Units

In this section, the Bendable Linear Unit (BLUs), a new parametric activation function, is introduced. BLUs have two parameters: α and β, where $0 \leq \alpha, \beta \leq 1$. The output of a BLU is

$$f(\alpha, \beta, x) = \beta\big(\sqrt{x^2 + \alpha^2 + \epsilon} - \alpha\big) + x. \tag{8.2}$$

α controls the sharpness of the function, interpolating between something like LReLU and SoftPlus [51]. When α is 0, BLU is sharp, like LReLU. When $0 < \alpha \leq 1$, BLU is smooth, like SoftPlus; higher values produce a smoother curve. By adding a small value inside the square root, ϵ, we avoid the discontinuity in the derivative when $x = \alpha = 0$. Figure 8.3 visualizes the effect of α on BLU.

β controls the shape of the function, interpolating between the identity function and a rectifier. When β is 0, BLU is exactly the identity function. When $0 < \beta \leq 1$, the

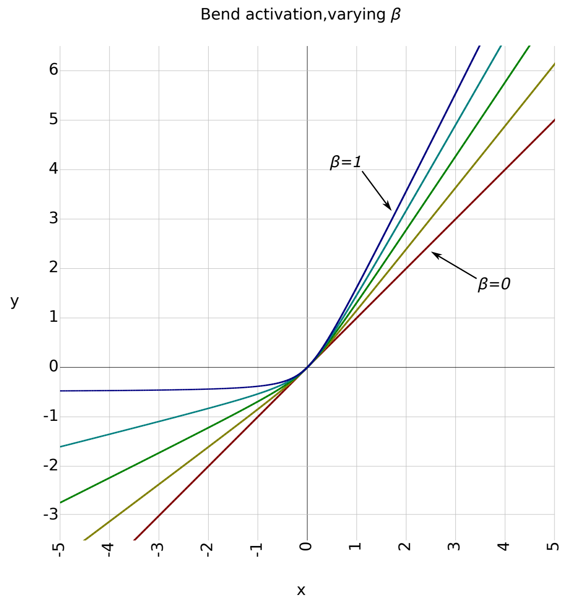

Figure 8.2: A visualization of Bendable Linear Units, with α fixed at 0.5. The closer β is to 0, the more it is like the identity function, and the closer β is to 1, the steeper the bend.

function produces a nonlinear bend; higher values produce a steeper bend. The effect of the β parameter is shown in Figure 8.3.

The derivatives of BLU with respect to its input and parameters are

$$\frac{\partial f}{\partial x} = \frac{\beta x}{\sqrt{x^2+\alpha^2+\epsilon}} + 1,$$
$$\frac{\partial f}{\partial \alpha} = \beta\left(\frac{\alpha}{\sqrt{x^2+\alpha^2+\epsilon}} - 1\right), \text{ and}$$
$$\frac{\partial f}{\partial \beta} = \sqrt{x^2+\alpha^2+\epsilon} - \alpha.$$

In a convolutional layer, the parameters of a BLU are shared so that there is only one of each per kernel, thus adding only 2 parameters per kernel to the network. In a fully connected layer, each unit has its own α and β. In Section 8.4, we also consider versions of BLU with one or both of its parameters fixed as constants, which further reduces the number of parameters required by BLU. We observe the following useful properties of BLUs.

First, **it contains the identity function**. This is one of the most important advantages of a BLU. Neural networks that use activation functions with this property can learn more effectively in deep networks than those that do not, for two reasons: 1) the gradient can neither explode nor vanish through the identity function, and 2) layers that are not needed to represent training data in a very deep network can be "skipped" using the identity function. These two reasons are precisely why residual connections have proven to be so useful in deep networks. Unintuitively, networks without access to the identity function tend to perform *worse* as they become deeper [147, 146, 66, 68]. Activation functions that can learn to become identity, like residual connections that use the identity function, enable deep networks to at least retain the same accuracy, if not improve accuracy in deeper networks. Because BLU contains identity, it can be utilized by deep networks to construct implicit residual connections. Thus, explicit residual connections are not required in networks using BLU as an activation (although residual connections are still useful in BLU networks; see Section 8.4). BLU has the particularly useful property of being identity when the parameters are zero, such that regularization of the parameters toward zero encourages the activation toward identity, which may help to prevent overfitting in deep networks (as is the case with APLUs [2] and residual connections [68]).

Second, **it is a non-saturating nonlinearity**. Perhaps the greatest disadvantage of saturating nonlinearities like hyperbolic tangent is that their derivatives asymptotically approach 0, with the result that training can slow down or even halt as the gradient approaches 0. Non-saturating nonlinearities, like BLU and ReLU, cannot get stuck in these regions of slow learning and are thus able to learn more quickly than saturating activation functions [50, 102, 115]. Except in the case where $\beta = 1$, the derivative of a BLU is non-zero for both positive and negative inputs, like LReLU, as opposed to vanilla ReLU, ELU, and PELU, the derivatives of which have regions of saturation for negative inputs. Although recent work has suggested that a good activation function should be saturating for negative values in order to dampen variance if it grows to large [96], we claim that in some cases it is useful to have an activation that does not saturate at all, as we demonstrate in Section 8.4. This claim is consistent with related works that use leaky or parametric ReLUs [115, 56, 67]. Furthermore, BLU has the adaptability to choose between saturating and non-saturating for negative inputs, synthesizing the benefits of PReLU, ELU, and scaled ELU (SELU) activations [96]. When the parameters of BLU *do* cause saturation, BLU captures the benefit of dampening variance as in ELU, thus avoiding exploding gradients in deep networks.

Third, **it can have a slope greater than 1 for positive inputs**. BLU can model the identity function, but it also has the capacity for growth for positive inputs; when β is 1, the positive-side slope of BLU is 2. The importance of this property has been discussed in previous work [96]. A slope greater than 1 for positive inputs avoids a vanishing gradient in deep networks. Combined with the capacity to saturate for negative inputs, this property enables BLU to avoid both exploding and vanishing gradients, leading to stabler training.

Fourth, **it is C^∞ continuous**. Every order of derivative is well-defined and smooth, in contrast with piecewise activations like ReLU and ELU and related functions. This is particularly useful in the first derivative, which yields a smoother error surface than nonlinearities that are not C^∞ continuous. Prior to the success of ReLUs, it was commonly held that activation functions *must* be infinitely differentiable [121, 179, 108, 151], which is also a common assumption made when dealing with extreme learning machines [77, 122]. Although this property is no longer commonly assumed or required for deep learning, the smoothness of C^∞ continuous activation functions can theoretically be exploited to speed up optimization [117, 167, 178].

8.4 Experiments

In this section, we present our experimental approach and the results of three experiments comparing parametric and non-parametric activation functions. All of our experiments use the CIFAR-10 dataset [101] and a Wide Residual Network (WRN) topology [172]. We selected WRNs as they have hyperparameters d and k to easily control the depth and width of the topology, respectively, so that we were able to test the effect of different activations for different network sizes. See Table 8.1 for a description of a WRN-d-k topology. For experiments with BLUs, we initialize α and β parameters to using a random uniform distribution. We used the Keras deep learning library [24] with the TensorFlow backend [1] for all tests. In our first experiment, we compare nine activation functions across five topologies in order to investigate the potential accuracy gains of parameterizing activation functions. In our second experiment, we compare nine activation functions using the WRN-16-4 topology trained for a short amount of time to test how quickly networks with each activation converge. In our third experiment, we evaluate the claim that parametric activations that can approximate the identity function are more robust when removing residual connections in deep networks. In all of our experiments, we use the CIFAR-10 task to compare three groups of activation functions:

1. piecewise linear units: ReLU, PReLU, and APLU,
2. exponential units: ELU and PELU, and
3. four variations of BLU:
    - BLU-$\alpha\beta$ (learned α and β),
    - BLU-α (learned α, fixed $\beta = 0.5$),
    - BLU-β (learned β, fixed $\alpha = 0.5$), and
    - BLU-const (fixed $\alpha = 0.5$ and $\beta = 0.5$).

Although some of these have been compared before (i.e. ReLU and PReLU, ELU and PELU), a full comparison of these methods across several topologies has not been done until now. The code that produced the reported results is available at the following URL:

`https://github.com/thelukester92/2018-blu/`.

In our first experiment, we train 45 models using the nine activation functions across five WRN topologies: WRN-40-1, WRN-40-2, WRN-40-4, WRN-16-4, and WRN-16-8. The

Table 8.1: The wide residual network (WRN) topology [172] used in some of our experiments. A WRN has two hyperparameters: d (which controls depth) and k (which controls width). n, used in the table, is defined as $\frac{d-4}{6}$. The final column, Number, is how many copies of the layer are used in succession.

Layer	**Stride**	**Number**
conv 16 x 3 x 3	(1, 1)	1
[conv 16k x 3 x 3]	(1, 1)	n
conv 32 x 3 x 3	(2, 2)	1
[conv 32k x 3 x 3]	(1, 1)	$n-1$
conv 64 x 3 x 3	(2, 2)	1
[conv 64k x 3 x 3]	(1, 1)	$n-1$
global max pool	-	1
fully connected 10	-	1
softmax	-	1

goal of this experiment is to compare final training accuracies of each activation, and to note how the accuracies are affected by network width and depth. Weights are initialized with LSUV initialization [124]. We train each model using stochastic gradient descent with a momentum term of 0.9, a weight decay of $5e^{-4}$, and a mini-batch size of 128. For the learning rate, we use cosine annealing [113] starting from $1e^{-2}$ and decaying to $1e^{-6}$ over the course of training (without restarting). Training runs for 300 epochs for each model. The results of this experiment are recorded in Table 8.2.

We find that in all but one case (WRN-40-1), PReLU outperforms ReLU at least marginally. Similarly, PELU achieves about the same or slightly better accuracy compared with ELU. Although the accuracy gained by the parameterized activations are marginal at best, it is clear that the additional parameters are in no significant way increasing the risk of overfitting. The results on the four variations of BLU are interesting. The most parameterized version, BLU-$\alpha\beta$, does not tend to perform as well as the other variations. This may be due to additional local optima introduced by both parameters, and the relationship between α and β. In the deep topologies (40 depth), BLU-β yields higher accuracies than the other versions of BLU. Aside from the very narrow WRN-40-1 topology, BLU-β outperforms all other compared activations. We argue that this is related to the fact that BLU-β, unlike BLU-α

Table 8.2: The final CIFAR-10 test set accuracy for each activation/topology pair tested in our first experiment. The best activation for each topology is shown in **bold**. Parametric activations tend to achieve higher accuracy than their non-parametric counterparts, although not always by a significant margin.

Activation	**WRN-40-1**	**WRN-40-2**	**WRN-40-4**	**WRN-16-4**	**WRN-16-8**
ReLU	92.30%	93.73%	94.29%	93.61%	93.21%
PReLU	92.16%	94.37%	95.05%	94.18%	94.71%
APLU	92.18%	93.88%	94.27%	94.43%	94.54%
ELU	**93.76%**	94.68%	95.20%	94.86%	**95.25%**
PELU	93.50%	94.68%	**95.36%**	**95.04%**	95.14%
BLU-$\alpha\beta$	93.30%	94.40%	95.09%	94.62%	94.86%
BLU-α	93.42%	94.86%	94.69%	94.66%	94.78%
BLU-β	93.51%	**95.02%**	94.54%	94.44%	94.71%
BLU-const	93.48%	94.83%	94.59%	94.60%	93.84%

and BLU-const, can model identity and thus can be used by the network to add something resembling residual connections. In fact, upon examination of the learned parameters of β, we find that many of the parameters indeed went to 0 to model identity, particularly in the lower layers. Figure 8.4 shows one of the β values in the lowest layer over time; the network learns to tune it from its starting point (about 0.2) to near 0. For the final two experiments, we exclusively use BLU-β as the preferred parameterization of BLU.

In our second experiment, we compare the effect of limited training time on each of the nine activation functions. We use the WRN-16-4 topology and the same initialization and training method as in our first experiment, but we limit the number of epochs from 300 to 100. The training curves of five of the activations (PELU, ELU, ReLU, PReLU, and BLU) are given in Figure 8.4. The final accuracies for all nine activations are given in Table 8.3. We note that the parametric activation functions, BLU, PELU, and PReLU, all learn more quickly than the non-parametric functions.

In our third experiment, we consider the effect of residual connections on parametric and non-parametric activations. We compare the top five activations from our first experiment: ReLU, PReLU, ELU, PELU, and BLU. For each activation, we use the WRN-40-4 topology with the residual connections removed. No changes were made to our training method from

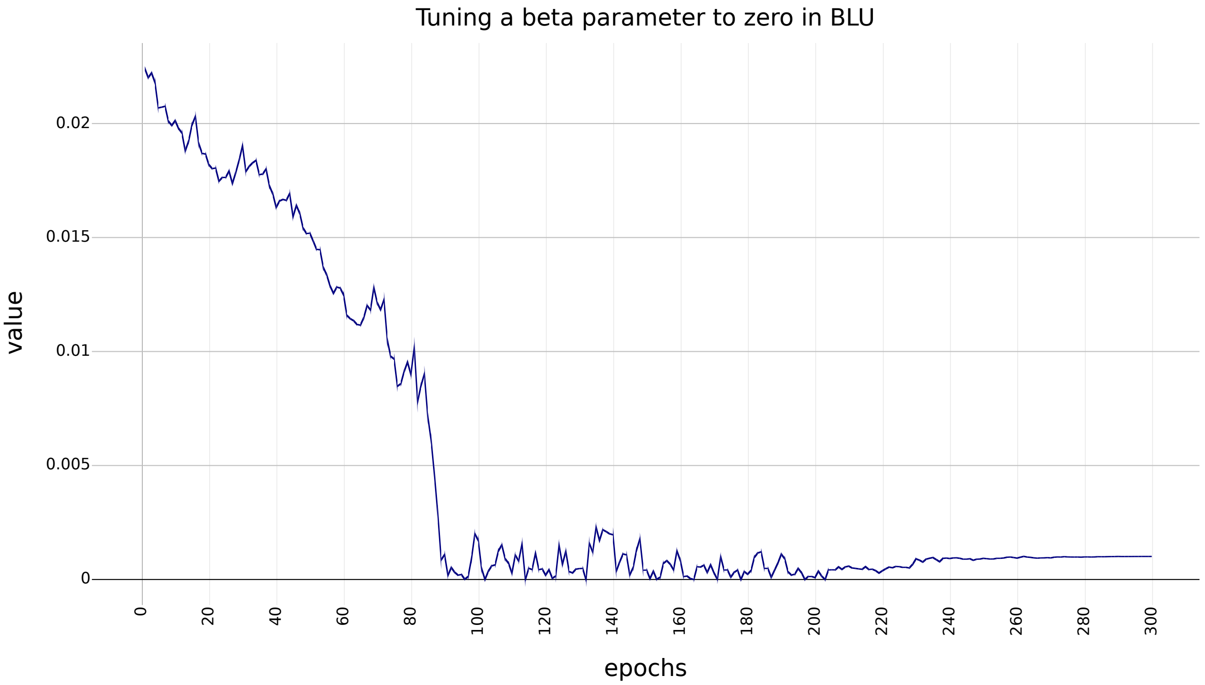

Figure 8.3: One of the β values from the lowest layer of the WRN-16-4 network using the BLU-β activation. The initial value, which is about 0.2, is selected from a uniform random distribution. During the course of training, the network tunes this value to almost 0, forming a kind of learned residual connection.

Table 8.3: The final CIFAR-10 test set accuracy for each activation using a WRN-16-4 topology in our second experiment, limiting training time to 100 epochs. The parametric activations learn faster than the non-parametric activations. The best result is shown in **bold**.

Activation	**Accuracy**
ReLU	91.19%
PReLU	92.26%
APLU	91.27%
ELU	93.47%
PELU	**93.49%**
BLU-$\alpha\beta$	92.90%
BLU-α	92.34%
BLU-β	93.19%
BLU-const	92.72%

the first experiment, except that residual conenctions had been removed from the topology. The final accuracy of these models is provided in Table 8.4. BLU performs the best, achieving 89.4% accuracy on the CIFAR-10 test set without residual connections and experiencing

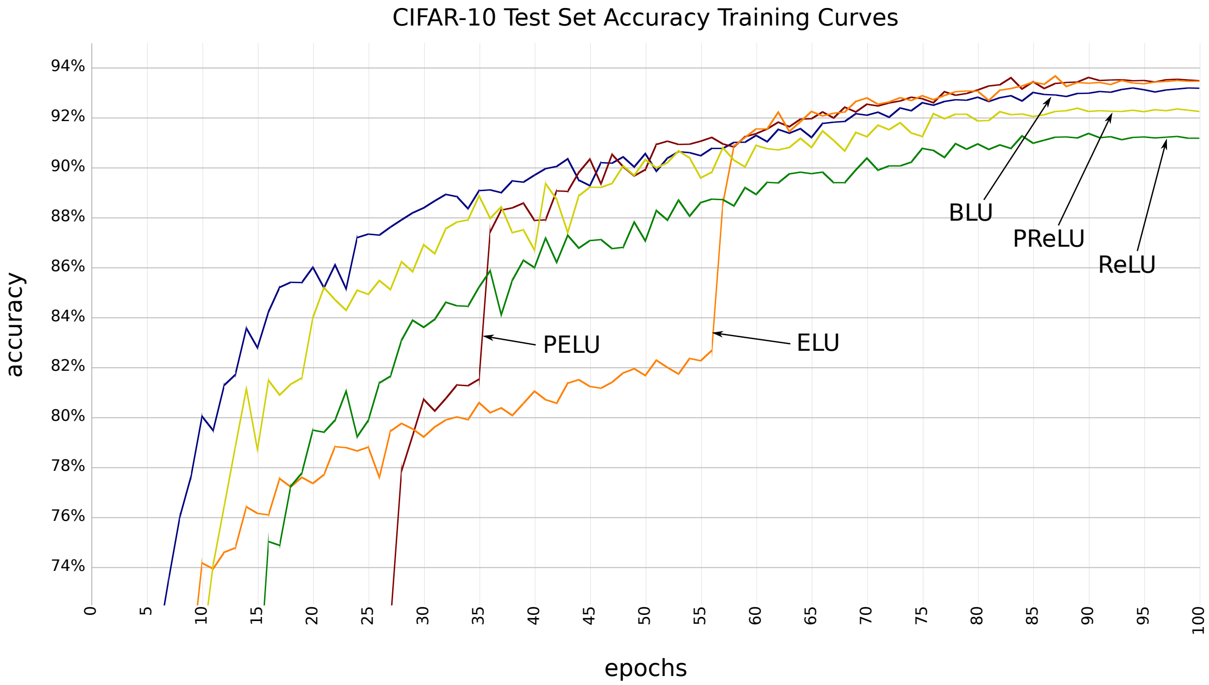

Figure 8.4: Selected training curves of test set accuaracy on the CIFAR-10 task from our first experiment. We include curves for BLU, ELU (in orange), PELU (in red), ReLU (in green), and PReLU (in yellow). BLU learns the fastest, but does not converge to as good a final accuracy as PELU and ELU. PELU and ELU achieve approximately the same final result, but PELU converges faster. PReLU, on the other hand, yields significantly better results than ReLU. Thus, for both PELU and PReLU, we observe a benefit to the parameterization.

only 5.14% absolute loss of accuracy. PELU completely diverged during training, even for different random seeds. ELU performed the second best by a narrow margin, suggesting that it is robust against residual connections by some other mechanism than the identity function. In terms of absolute loss compared with the first experiment, ELU experienced more loss than BLU did, with an absolute percent difference of 6.30%. As expected, PReLU vastly outperforms ReLU, achieving 236% relative improvement and nearly 50% absolute improvement over ReLU.

The good performance of ELU on this third experiment strengthens the claim that it is indeed useful to have an activation that saturates on one side, as has been previously suggested [96]. On the other hand, the massive improvement of PReLU over ReLU backs up the idea that it is useful to have an activation that can approximate the identity function

Table 8.4: The final CIFAR-10 test set accuracy for each activation tested in our third experiment (using WRN-40-4 with no residual connections). The difference in accuracy compared with WRN-40-4 *with* residual connections is reported in the last column. BLU performed the best in this experiment, both in terms of accuracy and difference in loss. PELU, marked with "x", diverged during training for several random seeds. ELU achieved good results, despite not being able to approximate the identity function. Though not as good as ELU or BLU, PReLU vastly outperforms ReLU. The best result (highest accuracy and lowest difference in accuracy) is shown in **bold**.

Activation	Accuracy	Difference
ReLU	35.30%	58.99%
PReLU	83.66%	11.39%
ELU	88.90%	6.30%
PELU	x	x
BLU	**89.40%**	**5.14%**

and is thus non-saturating on both sides [115, 56, 67]. The improvement of BLU over PReLU demonstrates that a slope greater than 1 (on the right side) is useful in an activation function [96]. The only activation function that synthesizes all three of these observations is BLU. It has the capacity to saturate on the left side or it can approximate identity, adaptively learning which one better fits the data. At the same time, BLU can learn to have linear growth on the right side or superlinear growth (quadratic). All of these properties come together in this third experiment to yield the highest accuracy without residual connections.

8.5 Conclusion

In this chapter, we considered the topic of parametric activation functions. We found that parameterizing activation functions in neural networks do not tend to overfit and tend to converge more quickly than non-parametric activations. We introduced the Bendable Linear Unit (BLU), which synthesizes many useful properties of other activations, including PReLU, ELU, and SELU. BLU was shown to outperform other activations when using a topology without residual connections. As deep learning advances, we desire to see further exploration of this subdomain of neural network nonlinearities as a method of constructing faster, more powerful models.

Part III

CONCLUSION

Chapter 9

Summary

In this dissertation, a number of neural network methods were presented which use the aggregation and parameterization of activation functions. We advance the state of heterogenous and parameterized activation functions, continuing in work started by previous researchers as well as exploring novel methods and applications in this subdomain of deep learning. We claim and demonstrate that the aggregation of heterogenous activation functions and the parameterization of activation functions can enable neural networks to model data more quickly and more accurately. Heterogenous activation functions are shown to be well-suited for modeling data sampled from the composition of heterogenous signals, and parametric activation functions are shown to promote simpler models, reduce overfitting, and generalize more effectively.

Three novel parametric activation functions were introduced in this work, including Soft Exponential (SoftExp), a fuzzy logic activation, and Bendable Linear Units (BLUs). SoftExp is a mathematically elegant interpolation between logarithm, linear, and exponential functions, and can be used in a neural network to exactly compute a number of common mathematical operations such as addition, multiplication, inner product, distance, polynomials, and sinusoids. The fuzzy logic activation function is the first activation to adaptively learn fuzzy logic operations by gradient-based optimization, and is a step toward neural network transparency. BLU synthesizes the beneficial properties of PReLU (it can become the identity function), ELU (it can saturate one side of the input space), and SELU (it can grow with magnitude greater than 1); BLU is shown to be more robust than other activations when removing residual connections in deep networks.

An evaluation of parametric activation functions and the utility of the parameterization is presented in this dissertation. Such a comparison had not previously been conducted at the level of activation functions. Our experiments demonstrate that parametric activations offer a representational capacity beyond non-parametric activations, achieving higher accuracy, converging more quickly, and being more robust against the loss of residual connections.

We continued exiting work with aggregating activation functions for time-series analysis using Neural Decomposition (ND). It is demonstated that parameters for sinusoidal activations can only effectively be learned by gradient descent by the addition of an augmentation function that captures linear and non-periodic, nonlinear components of a decomposable signal. Careful attention is made to the training process in this mixed model, noting that some activations are more sensitive to intialization, regularization, and input preprocessing. ND is shown to outperform other methods, including the widely used LSTM network, for some real-world datasets.

Finally, we built upon previous work that aggregates neuron inputs using product instead of sum in product unit neural networks (PUNNs). An approach is presented that applies a window to the product operation, creating windowed product unit neural networks (WPUNNs). WPUNNs solve the major problems associated with training PUNNs. This new method is demonstrated to be effective without traditional nonlinearities. Furthermore, WPUNNs can be used as a generalization of gated units in recurrent neural networks.

The works represented in this dissertation form a significant body of information that advances current knowledge regarding activation functions in neural networks. The aggregation of heterogenous activation functions is useful for signal decomposition, and the parameterization of activation functions has a variety of applications. Parametric activations can also be used to accelerate network convergence, speeding up learning and reducing training time. The combination of the presented methods can produce simpler, faster, and more accurate neural networks.

Chapter 10

Future Work

Activation functions play a critical role in deep learning, enabling neural networks to fit complex data in high dimensional space. In just the last decade, there has been considerable work dedicated to research in this field, particularly since the introduction of ReLU in 2010 [125]. The nonlinearities these works have explored range from adaptations of ReLU [115, 165, 67] to novel parametric activation functions [52, 158, 54]. Considering this trajectory and the research contained in this dissertation, we expect future work in this domain to further explore parametric activation functions. The increased representational capacity, model simplification, and acceleration of learning afforded by these parameterizations has the potential to lead to smaller, faster models that achieve the same accuracy as larger, slower models. This also has the potential to make deep learning more accessible, overcoming the high financial cost and time commitment required for training many existing models.

Some parametric activations generalize well enough that explicit aggregation may not be necessary; Soft Exponential, for example, can theoretically be used as the only activation function in an implementation of Neural Decomposition. Because of this, we expect a greater focus on parameterization than on aggregation, and existing work supporting aggregation also implicitly supports parameterization.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Forest Agostinelli, Matthew Hoffman, Peter Sadowski, and Pierre Baldi. Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*, 2014.

[3] Mohamed R Amer and Sinisa Todorovic. Sum-product networks for modeling activities with stochastic structure. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 1314–1321. IEEE, 2012.

[4] Mohamed R Amer and Sinisa Todorovic. Sum product networks for activity recognition. *IEEE transactions on pattern analysis and machine intelligence*, 38(4):800–813, 2016.

[5] A. Asuncion and D. J. Newman. UCI machine learning repository, 2007.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] Michael W Berry, Murray Brown, Amy N Langville, V Paul Pauca, and Robert J Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, 2007.

[8] Peter Bloomfield. *Fourier analysis of time series: an introduction*. John Wiley & Sons, 2004.

[9] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time Series Analysis: Forecasting and Control*. Wiley, June 2008.

[10] Sofiane Brahim-Belhouari and Amine Bermak. Gaussian process for nonstationary time series prediction. *Computational Statistics & Data Analysis*, 47(4):705–712, 2004.

[11] Jean Philippe Brunet, Pablo Tamayo, Todd R Golub, and Jill P Mesirov. Metagenes and molecular pattern discovery using matrix factorization. *Proceedings of the National Academy of Sciences*, 101(12):4164–4169, 2004.

[12] J.F. Cai, Emmanuel J. Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010.

[13] Juana Canul-Reich, Larry Shoemaker, and Lawrence O Hall. Ensembles of fuzzy classifiers. In *Fuzzy Systems Conference, 2007. FUZZ-IEEE 2007. IEEE International*, pages 1–6. IEEE, 2007.

[14] Gail A Carpenter, Stephen Grossberg, Natalya Markuzon, John H Reynolds, and David B Rosen. Fuzzy artmap: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on neural networks*, 3(5):698–713, 1992.

[15] Chris Chatfield. *Time-series forecasting*. CRC Press, 2000.

[16] Chris Chatfield. *The Analysis of Time Series: An Introduction*. Chapman and Hall/CRC, 6 edition, July 2003.

[17] CL Philip Chen, Yan-Jun Liu, and Guo-Xing Wen. Fuzzy neural network-based adaptive control for a class of uncertain nonlinear stochastic systems. *IEEE Transactions on Cybernetics*, 44(5):583–593, 2014.

[18] Sheng Chen, Colin FN Cowan, and Peter M Grant. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks*, 2(2):302–309, 1991.

[19] Shyi-Ming Chen. Forecasting enrollments based on fuzzy time series. *Fuzzy Sets and Systems*, 81(3):311–319, August 1996.

[20] SX Chen, HB Gooi, and MQ Wang. Solar radiation forecast based on fuzzy logic and neural networks. *Renewable Energy*, 60:195–201, 2013.

[21] Yie-Chien Chen and Ching-Cheng Teng. A model reference control structure using a fuzzy neural network. *Fuzzy sets and Systems*, 73(3):291–312, 1995.

[22] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[23] Tsan-Ming Choi, Yong Yu, and Kin-Fan Au. A hybrid sarima wavelet transform method for sales forecasting. *Decision Support Systems*, 51(1):130–140, 2011.

[24] François Chollet et al. Keras. `https://github.com/keras-team/keras`, 2015.

[25] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[26] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012.

[27] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[28] Earl Cox. Adaptive fuzzy systems. *IEEE Spectrum*, 30(2):27–31, 1993.

[29] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[30] Jan G De Gooijer and Rob J Hyndman. 25 years of time series forecasting. *International journal of forecasting*, 22(3):443–473, 2006.

[31] Georg Dorffner. Neural networks for time series processing. In *Neural Network World*. Citeseer, 1996.

[32] Harris Drucker, Chris J.C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. In *Advances in Neural Information Processing Systems 9*, 1996.

[33] Włodzisław Duch and Norbert Jankowski. Survey of neural transfer functions. *Neural Computing Surveys*, 2(1):163–212, 1999.

[34] Wlodzislaw Duch and Norbert Jankowski. Transfer functions: hidden possibilities for better neural networks. In *ESANN*, pages 81–94. Citeseer, 2001.

[35] AM Durán-Rosal, C Hervás-Martínez, AJ Tallón-Ballesteros, AC Martínez-Estudillo, and S Salcedo-Sanz. Massive missing data reconstruction in ocean buoys with evolutionary product unit neural networks. *Ocean Engineering*, 117:292–301, 2016.

[36] Richard Durbin and David E Rumelhart. Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural computation*, 1(1):133–142, 1989.

[37] Christian E. Elger and Klaus Lehnertz. Seizure prediction by non-linear time series analysis of brain electrical activity. *European Journal of Neuroscience*, 10(2):786–789, February 1998.

[38] Andries P. Engelbrecht, Ap Engelbrecht, and A Ismail. Training product unit neural networks, 1999.

[39] Katya Rubia et al. Hypofrontality in attention deficit hyperactivity disorder during higher-order motor control: a study with functional mri. *American Journal of Psychiatry*, 1999.

[40] Yuchen Fan, Yao Qian, Feng-Long Xie, and Frank K Soong. Tts synthesis with bidirectional lstm based recurrent neural networks. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[41] F. Fernández-Navarro, Maria Angeles de la Cruz, P. A. Gutiérrez, A. Castaño, and C. Hervás-Martínez. Time series forecasting by recurrent product unit neural networks. *Neural Computing and Applications*, Jul 2016.

[42] Ray J Frank, Neil Davey, and Stephen P Hunt. Time series prediction and neural networks. *Journal of Intelligent and Robotic Systems*, 31(1-3):91–103, 2001.

[43] M. S. Gashler. Waffles: A machine learning toolkit. *Journal of Machine Learning Research*, 12:2383–2387, July 2011.

[44] Michael S Gashler and Stephen C Ashmore. Training deep fourier neural networks to fit time-series data. *Lecture Notes in Bioinformatics*, 8590:48–55, 2014.

[45] Michael S. Gashler and Stephen C. Ashmore. Training deep fourier neural networks to fit time-series data. In *Intelligent Computing in Bioinformatics - 10th International Conference, ICIC 2014, Taiyuan, China, August 3-6, 2014. Proceedings*, pages 44–55, 2014.

[46] Felix A Gers, Douglas Eck, and Jürgen Schmidhuber. Applying lstm to time series predictable through time-window approaches. In *Artificial Neural Networks—ICANN 2001*, pages 669–676. Springer, 2001.

[47] Felix A Gers and E Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[48] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[49] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[50] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[51] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[52] Luke B Godfrey and Michael S Gashler. A continuum among logarithmic, linear, and exponential functions, and its potential to improve generalization in neural networks. In *Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K), 2015 7th International Joint Conference on*, volume 1, pages 481–486. IEEE, 2015.

[53] Luke B Godfrey and Michael S Gashler. Neural decomposition of time-series data for effective generalization. *IEEE transactions on neural networks and learning systems*, 2017.

[54] Luke B Godfrey and Michael S Gashler. A parameterized activation function for learning fuzzy logic operations in deep neural networks. In *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*. IEEE, 2017.

[55] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[56] Benjamin Graham. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070*, 2014.

[57] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[58] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 273–278. IEEE, 2013.

[59] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

[60] Alain Guerrero-Enamorado and Daimerys Ceballos-Gastell. An experimental study of evolutionary product-unit neural network algorithm. *Computación y Sistemas*, 20(2), 2016.

[61] Madan M Gupta and J Qi. Theory of t-norms and fuzzy inference methods. *Fuzzy sets and systems*, 40(3):431–450, 1991.

[62] PA Gutiérrez, F López-Granados, JM Peña-Barragán, M Jurado-Expósito, and C Hervás-Martínez. Logistic regression product-unit neural networks for mapping ridolfia segetum infestations in sunflower crop using multitemporal remote sensed data. *Computers and Electronics in Agriculture*, 64(2):293–306, 2008.

[63] PA Gutiérrez, MJ Segovia-Vargas, S Salcedo-Sanz, C Hervás-Martínez, A Sanchis, JAea Portilla-Figueras, and F Fernández-Navarro. Hybridizing logistic regression with product unit and rbf networks for accurate detection and prediction of banking crises. *Omega*, 38(5):333–344, 2010.

[64] Pedro Antonio Gutiérrez, César Hervás, M Carbonero, and Juan Carlos Fernández. Combined projection and kernel basis functions for classification in evolutionary neural networks. *Neurocomputing*, 72(13):2731–2742, 2009.

[65] Yoichi Hayashi, James J Buckley, and Ernest Czogala. Fuzzy neural network with fuzzy signals and weights. *International Journal of Intelligent Systems*, 8(4):527–537, 1993.

[66] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*, pages 5353–5360. IEEE, 2015.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[68] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[69] Robert. Hecht-Nielsen. Theory of the backpropagation neural network. *International Joint Conference on Neural Networks*, 1989.

[70] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[71] César Hervás, Francisco J Martínez, and Pedro Antonio Gutiérrez. Classification by means of evolutionary product-unit neural networks. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, pages 1525–1532. IEEE, 2006.

[72] César Hervás-Martínez and Francisco Martínez-Estudillo. Logistic regression using covariates obtained by product-unit neural network models. *Pattern Recognition*, 40(1):52–64, 2007.

[73] Keith W. Hipel and A. I. McLeod. *Time Series Modelling of Water Resources and Environmental Systems*. Elsevier, January 1994.

[74] Keith W Hipel and A Ian McLeod. *Time series modelling of water resources and environmental systems*, volume 45. Elsevier, 1994.

[75] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[76] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[77] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[78] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[79] Adiel Ismail and Andries Petrus Engelbrecht. Global optimization algorithms for training product unit neural networks. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 1, pages 132–137. IEEE, 2000.

[80] Kyoung jae Kim. Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1-2):307–319, September 2003.

[81] Herbert Jaeger. Reservoir riddles: Suggestions for echo state network research. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 3, pages 1460–1462. IEEE, 2005.

[82] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

[83] J-SR Jang. Anfis: adaptive-network-based fuzzy inference system. *IEEE transactions on systems, man, and cybernetics*, 23(3):665–685, 1993.

[84] J-SR Jang and C-T Sun. Functional equivalence between radial basis function networks and fuzzy inference systems. *IEEE transactions on Neural Networks*, 4(1):156–159, 1993.

[85] Jyh-Shing Roger Jang et al. Fuzzy modeling using generalized neural networks and kalman filter algorithm. In *AAAI*, volume 91, pages 762–767, 1991.

[86] Jyh-Shing Roger Jang, Chuen-Tsai Sun, and Eiji Mizutani. Neuro-fuzzy and soft computing, a computational approach to learning and machine intelligence. 1997.

[87] David J Janson and James F Frenzel. Training product unit neural networks with genetic algorithms. *IEEE Expert*, 8(5):26–33, 1993.

[88] Abdul Salam Jarrah, Reinhard Laubenbacher, Brandilyn Stigler, and Michael Stillman. Reverse-engineering of polynomial dynamical systems. *Advances in Applied Mathematics*, 39(4):477–489, 2007.

[89] Ma Jun and Meng Ying. Research of traffic flow forecasting based on neural network. In *Intelligent Information Technology Application, 2008. IITA '08. Second International Symposium on*, volume 2, pages 104–108, December 2008.

[90] Iebeling Kaastra and Milton Boyd. Designing a neural network for forecasting financial and economic time series. *Neurocomputing*, 10(3):215–236, 1996.

[91] Barry L Kalman and Stan C Kwasny. Why tanh: choosing a sigmoidal function. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 4, pages 578–581. IEEE, 1992.

[92] DP Kanungo, MK Arora, S Sarkar, and RP Gupta. A comparative study of conventional, ann black box, fuzzy and combined neural and fuzzy weighting procedures for landslide susceptibility zonation in darjeeling himalayas. *Engineering Geology*, 85(3):347–366, 2006.

[93] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[94] Erkan Kayacan, Erdal Kayacan, and Mojtaba Ahmadieh Khanesar. Identification of nonlinear dynamic systems using type-2 fuzzy neural networksa novel learning algorithm and a comparative study. *IEEE Transactions on Industrial Electronics*, 62(3):1716–1724, 2015.

[95] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[96] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pages 972–981, 2017.

[97] George Klir and Bo Yuan. *Fuzzy sets and fuzzy logic*, volume 4. Prentice hall New Jersey, 1995.

[98] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[99] Bart Kosko. Neural networks and fuzzy systems: a dynamical systems approach to machine intelligence/book and disk. *Vol. 1Prentice hall*, 1992.

[100] Viktoriya Krakovna and Moshe Looks. A minimalistic approach to sum-product network learning for real applications. *arXiv preprint arXiv:1602.04259*, 2016.

[101] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[102] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

[103] Ren Jie Kuo, CH Chen, and YC Hwang. An intelligent stock trading decision support system through integration of genetic algorithm based fuzzy neural network and artificial neural network. *Fuzzy sets and systems*, 118(1):21–45, 2001.

[104] Hon Keung Kwan and Yaling Cai. A fuzzy neural network and its application to pattern recognition. *IEEE transactions on Fuzzy Systems*, 2(3):185–193, 1994.

[105] Chuen-Chien Lee. Fuzzy logic in control systems: fuzzy logic controller. i. *IEEE Transactions on systems, man, and cybernetics*, 20(2):404–418, 1990.

[106] Decai Li, Min Han, and Jun Wang. Chaotic time series prediction based on a novel robust echo state network. *Neural Networks and Learning Systems, IEEE Transactions on*, 23(5):787–799, 2012.

[107] Weizhuo Li. Combining sum-product network and noisy-or model for ontology matching. In *OM*, pages 35–39, 2015.

[108] Nan-Ying Liang, Guang-Bin Huang, Paramasivan Saratchandran, and Narasimhan Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Transactions on neural networks*, 17(6):1411–1423, 2006.

[109] C-T Lin and C. S. George Lee. Neural-network-based fuzzy logic control and decision system. *IEEE Transactions on computers*, 40(12):1320–1336, 1991.

[110] Chin-Teng Lin and CS George Lee. Reinforcement structure/parameter learning for neural-network-based fuzzy logic control systems. *IEEE Transactions on Fuzzy Systems*, 2(1):46–63, 1994.

[111] Chin-Teng Lin and CS George Lee. Neural fuzzy systems. *PTR Prentice Hall*, 1996.

[112] Marco Lippi, Matteo Bertini, and Paolo Frasconi. Short-term traffic flow forecasting: An experimental comparison of time-series analysis and supervised learning. In *IEEE Transactions on Intelligent Transportation Systems*, volume 14, pages 871–882, March 2013.

[113] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. 2016.

[114] Mantas Lukoševičius. *A practical guide to applying echo state networks*, volume 7700 of *Lecture Notes in Computer Science*, pages 659–686. Springer Berlin Heidelberg, 2 edition, 2012.

[115] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[116] Iain L MacDonald and Walter Zucchini. *Hidden Markov and other models for discrete-valued time series*, volume 110. CRC Press, 1997.

[117] Mehrdad Mahdavi, Lijun Zhang, and Rong Jin. Mixed optimization for smooth functions. In *Advances in Neural Information Processing Systems*, pages 674–682, 2013.

[118] Thomas Martinetz, Klaus Schulten, et al. A” neural-gas” network learns topologies. 1991.

[119] Francisco J Martínez-Estudillo, César Hervás-Martínez, Pedro Antonio Gutiérrez, and Alfonso C Martínez-Estudillo. Evolutionary product-unit neural networks classifiers. *Neurocomputing*, 72(1):548–561, 2008.

[120] Mazen Melibari, Pascal Poupart, Prashant Doshi, and George Trimponias. Dynamic sum product networks for tractable inference on sequence data. In *Conference on Probabilistic Graphical Models*, pages 345–355, 2016.

[121] Hrushikesh Narhar Mhaskar and Charles A Micchelli. How to choose an activation function. In *Advances in Neural Information Processing Systems*, pages 319–326, 1994.

[122] Yoan Miche, Antti Sorjamaa, Patrick Bas, Olli Simula, Christian Jutten, and Amaury Lendasse. Op-elm: optimally pruned extreme learning machine. *IEEE transactions on neural networks*, 21(1):158–162, 2010.

[123] Keiichiro Minami, Hiroshi Nakajima, and Takeshi Toyoshima. Real-time discrimination of ventricular tachyarrhythmia with fourier-transform neural network. *IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING*, 46(2), February 1999.

[124] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[125] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[126] O. Nerrand, P. Roussel-Ragot, D. Urbani, L. Personnaz, and G. Dreyfus. Training recurrent neural networks: Why and how ? an illustration in dynamical process modeling., 1994.

[127] Robert Peharz, Georg Kapeller, Pejman Mowlaee, and Franz Pernkopf. Modeling speech with sum-product networks: Application to bandwidth extension. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 3699–3703. IEEE, 2014.

[128] Mary L. Phillips. A differential neural response to threatening and non-threatening negative facial expressions in paranoid and non-paranoid schizophrenics. *Psychiatry Research: Neuroimaging*, 92(1):11–31, 1999.

[129] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 689–690. IEEE, 2011.

[130] Sultan Noman Qasem and Siti Mariyam Shamsuddin. Radial basis function network based on time variant multi-objective particle swarm optimization for medical diseases diagnosis. *Applied Soft Computing*, 11(1):1427–1438, 2011.

[131] Barry G. Quinn. Estimating the number of terms in a sinusoidal regression. *Journal of time series analysis*, 10(1):71–75, 1989.

[132] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[133] MaríA Dolores Redel-MacíAs, Francisco Fernández-Navarro, Pedro Antonio Gutiérrez, Antonio José Cubero-Atienza, and CéSar HerváS-MartíNez. Ensembles of evolutionary product unit or rbf neural networks for the identification of sound for pass-by noise test in vehicles. *Neurocomputing*, 109:56–65, 2013.

[134] Patrick E Rodi. On using genetic algorithm optimized activation functions to increase neural network accuracy. In *15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 3144, 2014.

[135] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[136] Simone Scardapane, Michele Scarpiniti, Danilo Comminiello, and Aurelio Uncini. Learning activation functions from data using cubic spline interpolation. *arXiv preprint arXiv:1605.05509*, 2016.

[137] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010.

[138] Bernhard Schölkopf, Kah-Kay Sung, Chris JC Burges, Federico Girosi, Partha Niyogi, Tomaso Poggio, and Vladimir Vapnik. Comparing support vector machines with gaussian kernels to radial basis function classifiers. *Signal Processing, IEEE Transactions on*, 45(11):2758–2765, 1997.

[139] Bruno Massoni Sguerra and Fabio G Cozman. Image classification using sum-product networks for autonomous flight of micro aerial vehicles. In *Intelligent Systems (BRACIS), 2016 5th Brazilian Conference on*, pages 139–144. IEEE, 2016.

[140] Zhiwei Shi and Min Han. Support vector echo-state machine for chaotic time-series prediction. *Neural Networks, IEEE Transactions on*, 18(2):359–372, 2007.

[141] Adrian Silvescu. Fourier neural networks. In *Neural Networks, 1999. IJCNN'99. International Joint Conference on*, volume 1, pages 488–491. IEEE, 1999.

[142] Adrian Silvescu. Fourier neural networks. In *Neural Networks, 1999. IJCNN '99. International Joint Conference on*, volume 1, pages 488–491, 1999.

[143] Ashish Sinha, Gayatri Kathayat, Hai Cheng, Sebastian F. M. Breitenbach, Max Berkelhammer, Manfred Mudelsee, Jayant Biswas, and R. L. Edwards. Trends and oscillations in the indian summer monsoon rainfall over the last two millennia. *Nat Commun*, 6, 02 2015.

[144] Jonas Sjöberg, Qinghua Zhang, Lennart Ljung, Albert Benveniste, Bernard Delyon, Pierre-Yves Glorennec, Håkan Hjalmarsson, and Anatoli Juditsky. Nonlinear black-box modeling in system identification: a unified overview. *Automatica*, 31(12):1691–1724, 1995.

[145] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[146] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[147] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[148] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[149] Elias M. Stein and Timothy S. Murphy. *Harmonic analysis: real-variable methods, orthogonality, and oscillatory integrals*, volume 3. Princeton University Press, 1993.

[150] Haowei Su, Ling Zhang, and Shu Yu. Short-term traffic flow prediction based on incremental support vector regression. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 1, pages 640–645, August 2007.

[151] Pawel Swietojanski, Jinyu Li, and Jui-Ting Huang. Investigation of maxout networks for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 7649–7653. IEEE, 2014.

[152] Antonio J Tallón-Ballesteros, César Hervás-Martínez, José C Riquelme, and Roberto Ruiz. Improving the accuracy of a two-stage algorithm in evolutionary product unit neural networks for classification by means of feature selection. In *International Work-Conference on the Interplay Between Natural and Artificial Computation*, pages 381–390. Springer, 2011.

[153] Antonio J Tallón-Ballesteros, César Hervás-Martínez, José C Riquelme, and Roberto Ruiz. Feature selection to enhance a two-stage evolutionary algorithm in product unit neural networks for complex classification problems. *Neurocomputing*, 114:107–117, 2013.

[154] HS Tan. Fourier neural networks and generalized single hidden layer networks in aircraft engine fault diagnostics. *Journal of engineering for gas turbines and power*, 128(4):773–782, 2006.

[155] Francis E.H. Tay and Lijuan Cao. Application of support vector machines in financial time series forecasting. *Omega*, 29(4):309–317, August 2001.

[156] James W. Taylor, Patrick E. McSharry, and Roberto Buizza. Wind power density forecasting using ensemble predictions and time series models. *Energy Conversion, IEEE Transactions on*, 24(3):775–782, September 2009.

[157] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 2012.

[158] Ludovic Trottier, Philippe Giguere, and Brahim Chaib-draa. Parametric exponential linear unit for deep convolutional neural networks. 2017.

[159] Andrew James Turner and Julian Francis Miller. Neuroevolution: evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, 7(3):135–154, 2014.

[160] Fevrier Valdez, Patricia Melin, and Oscar Castillo. An improved evolutionary method with fuzzy logic for combining particle swarm optimization and genetic algorithms. *Applied Soft Computing*, 11(2):2625–2632, 2011.

[161] Frans Van Den Bergh and Andries P Engelbrecht. Training product unit networks using cooperative particle swarm optimisers. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 1, pages 126–131. IEEE, 2001.

[162] Petr Vanicek. Approximate spectral analysis by least-squares fit. *Astrophysics and Space Science*, 4(4):387–391, 1969.

[163] Jung-Hua Wang, Yi-Wei Yu, and Jia-Horng Tsai. On the internal representations of product units. *Neural processing letters*, 12(3):247–254, 2000.

[164] William Wu-Shyong Wei. *Time series analysis*. Addison-Wesley publ, 1994.

[165] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[166] Wei Xu, Xin Liu, and Yihong Gong. Document clustering based on non-negative matrix factorization. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 267–273. ACM, 2003.

[167] Tianbao Yang. Trading computation for communication: Distributed stochastic dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 629–637, 2013.

[168] Y. Yokouchi. A strong source of methyl chloride to the atmosphere from tropical coastal land. *Nature*, 2000.

[169] Takuya Yoshioka, Katsunori Ohnishi, Fuming Fang, and Tomohiro Nakatani. Noise robust speech recognition using recent developments in neural networks for computer vision. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 5730–5734. IEEE, 2016.

[170] Lotfi A Zadeh. Fuzzy logic= computing with words. *IEEE transactions on fuzzy systems*, 4(2):103–111, 1996.

[171] Lotfi Asker Zadeh. Fuzzy logic. *Computer*, 21(4):83–93, 1988.

[172] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[173] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[174] Matthew D Zeiler, Marc'Aurelio Ranzato, Rajat Monga, Min Mao, Kun Yang, Quoc Viet Le, Patrick Nguyen, Alan Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3517–3521. IEEE, 2013.

[175] G Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, 2003.

[176] Guoqiang Zhang, B Eddy Patuwo, and Michael Y Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.

[177] Guoqiang Peter Zhang. Time series forecasting using a hybrid arima and neural network model. *Neurocomputing*, 50:159–175, January 2003.

[178] Lijun Zhang, Tianbao Yang, Rong Jin, and Xiaofei He. O (logt) projections for stochastic optimization of smooth and strongly convex functions. In *International Conference on Machine Learning*, pages 1121–1129, 2013.

[179] Xiao-Ping Zhang. Thresholding neural network for adaptive noise reduction. *IEEE Transactions on Neural Networks*, 12(3):567–584, 2001.

[180] Ying Zhang, Mohammad Pezeshki, Philémon Brakel, Saizheng Zhang, Cesar Laurent Yoshua Bengio, and Aaron Courville. Towards end-to-end speech recognition with deep convolutional neural networks. *arXiv preprint arXiv:1701.02720*, 2017.

[181] He Zhenya, Wei Chengjian, Yang Luxi, Gao Xiqi, Yao Susu, Russell C Eberhart, and Yuhui Shi. Extracting rules from fuzzy neural network by particle swarm optimisation. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 74–77. IEEE, 1998.

[182] Wei Zuo, Yang Zhu, and Lilong Cai. Fourier-neural-network-based learning control for a class of nonlinear systems with flexible components. *IEEE Transactions on Neural Networks*, 2009.